Automatic Goal Distribution Strategies for the Execution of Committed Choice Logic Languages on Distributed Memory Parallel Computers

Robert B. Scott

PhD
University of Edinburgh
1994

Abstract

There has been much research interest in efficient implementations of the Committed Choice Non-Deterministic (CCND) logic languages on parallel computers. To take full advantage of the speed gains of parallel computers, methods need to be found to automatically distribute goals over the machine processors, ideally with as little involvement from the user as possible.

In this thesis we explore some automatic goal distribution strategies for the execution of the CCND languages on commercially available distributed memory parallel computers.

There are two facets to the goal distribution strategies we have chosen to explore:

demand driven An idle processor requests work from other processors. We describe two strategies in this class: one in which an idle processor asks only neighbouring processors for spare work, the *nearest-neighbour* strategy; and one where an idle processor may ask any other processor in the machine for spare work, the *all-processors* strategy.

weights Using a program analysis technique devised by Tick, weights are attached to goals; the weights can be used to order the goals so that they can be executed and distributed out in weighted order, possibly increasing performance.

We describe a framework in which to implement and analyse goal distribution strategies, and then go on to describe experiments with demand driven strategies, both with and without weights. The experiments were made using two of our own implementations of Flat Guarded Horn Clauses — an interpreter and a WAM-like system — executing on a MEIKO T800 Transputer Array configured in a 2-D mesh topology.

Analysis of the results show that the all-processors strategies are promising (AP-NW), adding weights had little positive effect on performance, and that nearest-neighbours strategies can reduce performance due to bad load balancing.

We also describe some preliminary experiments for a variant of the AP-NW strategy: goals which suspend on one variable are sent to the processor that controls that variable, the *processes-to-data* strategy. And we briefly look at some preliminary results of executing programs on large numbers of processors (> 30).

Acknowledgments

I would like to give special thanks to my supervisors Chris Mellish and Murray Cole. I thank them for their discussions, suggestions and encouragement.

I would like to thank Paul Wilk for starting me on this enterprise.

I would also like to thank the many friends and colleagues who have provided a pleasant atmosphere in which to work. In no particular order I would like to thank especially: Andy Bowles, Rajiv Trehan, Brian Ross, Flavio Soares Correa Da Silva, Wamberto Vasconcelos, Nelson Ludlow, Carla Gomes, Keiichi Nakata, Ian Frank, Khee-Yin How, Jussi Stader, Sjoukje Osinga, David Goldsborough, Ian Lewin, and Tim Duncan.

Evelyn van de Veen deserves a special mention for her unbounded patience and happiness which sustained me through the dark periods of my PhD study.

Finally I would like to thank my parents, William and Kathleen Scott, for their unquestioning support, both spiritual and financial.

Declaration

I declare that this thesis has been composed by myself and that the research reported therein has been conducted by myself unless otherwise indicated.

Robert B. Scott

Contents

List of Figures

List of Tables

Monster!

Chapter 1

Introduction

1.1 Motivation

The general area of developing and implementing parallel logic programming languages has several motivations: the desire to exploit the speed of parallel computers and the desire for a good programming environment in which to develop programs on these computers. In the field of Artificial Intelligence (AI) these motivations are very important: many AI problems involving very large computations become more tractable on parallel machines and developing large systems requires a good programming environment.

To these ends researchers have developed and implemented parallel execution models of declarative computer languages based on functional programming and logic programming languages.

The committed-choice non-deterministic (CCND) logic languages have been specially designed to exploit the concept of concurrency so that parallel algorithms can be expressed as logic programs. Researchers have been looking for efficient methods of implementing these languages over a wide variety of computer types — sequential and parallel.

In this thesis we investigate one aspect of implementing these languages; that is, the problem of distributing work across the different processors with a view to minimising the overall execution time.

The basic unit of work (or process) is the reduction of a goal to some subgoals. A feature of the committed-choice logic languages is that goals are created dynamically and pass values between each other by binding shared logical variables. Execution of a committed-choice logic program, therefore, often leads to the creation of many small

processes that have complex dependencies between them and these dependencies can change dynamically during execution. This makes distributing goals over the processors of a distributed memory machine a complex problem.

The approach we have taken is to investigate one area of possible goal distribution strategies: automatic dynamic goal distribution strategies. **Automatic** means that the strategy tries to distribute goals with as little guidance from the user as possible; **dynamic** means that the decision of which processor to place a process on is made during program execution rather than beforehand.

In addition to being automatic and dynamic, we have in mind several criteria when choosing goal distribution strategies:

Distributed Each processor should operate an identical goal distribution strategy. This is to avoid bottlenecks that can occur when one processor has more responsibility for scheduling and distributing goals than others.

Scalable The distribution strategy should be implementable whether the computer has just 2 processors or has 2000 processors.

Low overhead Any goal distribution strategy will incur overheads. The overheads should be kept as small as possible.

Full parallelism The strategy should be able to support the parallelism of the language and not impose any restrictions.

A further prime aim of this work is to evaluate the performance of strategies using language implementations designed for execution on commercially available distributed memory parallel computers. To this end we designed and implemented two versions of the committed-choice language Flat Guarded Horn Clauses (FGHC) — an interpreter and an emulator. The goal distribution strategies were evaluated by incorporating them into our FGHC systems on which test programs were executed, with the systems running on a Meiko T800 Computing Surface distributed memory computer. The results of these executions were then analysed and evaluated according to measures which we have devised for that purpose.

1.2 Thesis layout

Here we describe the structure of the thesis:

Chapter 2 provides a context for the thesis by describing preliminary background information — the sorts of parallelism in logic languages, the syntax and seman-

tics of the committed choice languages, the organisation of parallel computer hardware — and by describing the main papers in the field of goal distribution strategies for the committed choice languages.

Chapter 3 is a detailed proposal of the experiments performed with goal distribution strategies. We describe a processing agent model for the design of a distributed language system and detail an interface between the processing agent and the goal queue of the agent. This interface becomes the basis for describing the goal distribution strategies that are used in experiments.

Also described here is the hardware on which the system executes, the test programs used for evaluation, and the procedure followed for the experiments.

An important part of this chapter is a description of the sorts of performance information that are gathered from an execution of the system and definitions of the performance measures used to compare the goal distribution strategies.

Chapter 4 describes our design and implementation of an interpreter for Flat Guarded Horn Clauses (FGHC), a language. The design is made for an array of T800 Transputers configured as a 2-D mesh topology. An evaluation of the performance of the system is given.

Chapter 5 details the results of the experiments made with the interpreter for the various goal distribution strategies, test programs, and varying numbers of processors. The results indicated that inefficiencies in the interpreter implementation may have led to disappointing results. We therefore decided to repeat the experiments using an emulator based system.

Chapter 6 describes the design and implementation of a WAM-like emulator derived from the interpreter described in chapter 4.

Chapter 7 details the results of the experiments made with the emulator in a similar manner to chapter 5. A detailed comparison is made between the results obtained from the interpreter and emulator experiments.

Chapter 8 begins with a description of preliminary experiments made with a variation on the goal distribution schedulers used in previous experiments. Another set of preliminary experiments conducted on large numbers of processors (between 36 and 100) is also described.

This chapter ends with directions for further work.

Chapter 9 begins with a summary of the contributions made by the work described in the thesis.

Finally, a summary of the conclusions of the experiments described in this thesis is presented.

Chapter 2

Background

2.1 Introduction

Sections in this chapter fall into two parts: preliminary background concepts, that are introduced to put the ideas in the thesis in context; and a survey of the current approaches to goal distribution for the committed-choice languages.

2.2 Prolog and parallelism

Here we give a brief description of execution models for Prolog in order to introduce the notion of parallel execution of logic languages and as a precursor to a description of the committed-choice logic languages. It is assumed that the reader has knowledge of Prolog and its usual sequential execution mechanism (see [CM87]).

Most sequential Prolog implementations use a left-to-right depth-first-with-backtracking computation rule — a form of execution that fits stack based sequential computers extremely well.

Much research effort has been put into parallel execution models for Prolog. On the surface it would seem that Prolog has excellent scope for parallelism:

OR-parallelism alternative clauses in a relation may be searched in parallel;

AND-parallelism goals in the body of a clause may be executed in parallel.

The problem with these approaches has been the extra implementation machinery necessary to exploit these forms of parallelism.

With OR-parallelism, the environment of the executing goal needs to be copied for each new clause branch of the relation executing in parallel. This is because global variables may become bound in each execution branch but only one binding can be current at any one time.

```
%program

  fruit(apple).
  fruit(banana).
  fruit(orange).

%query

  ?- fruit(X).
```

Figure 2.1: Example of OR-parallelism in Prolog

The Prolog program and query in figure 2.1 illustrates this problem. OR-parallel execution of `fruit(X)` results in the three clauses of *fruit/1* attempting to bind `X` with their respective fruit names. Some mechanism must be provided to make sure that `X` is bound to only one value at a time and that the correct order of bindings is preserved on backtracking if *fruit/1* should subsequently be failed.

With AND-parallelism, the problem is how to combine variable substitutions from goals that share variables and are executing in parallel.

```
%program

  fruit(apple).          veg(carrot).
  fruit(banana).         veg(tomato).
  fruit(tomato).         veg(pea).
  fruit(pear).           veg(cabbage).
  fruit(pea).

%query

  ?- fruit(X), veg(X).
```

Figure 2.2: Example of AND-parallelism in Prolog

The Prolog program and query in figure 2.2 illustrates this problem. AND-parallel execution of the query "`?- fruit(X),veg(X).`" will result in bindings for `X` from `fruit(X)` and from `veg(X)`. The problem is how to combine the resulting bindings

of the two goals while still preserving the usual Prolog order in which bindings are made and at the same time provide for backtracking if either of the goals should be subsequently failed.

One way of alleviating some of the problems associated with implementing AND-parallelism is to restrict the concurrent execution of a goal to those which do not share variables. Goal that do share variables can be executed in the usual sequential manner. In this way, the problems of collating the bindings of a variable shared between two goals are removed. Imposing this restriction gives rise to **restricted** or **independent AND-parallelism**[DeG84]. Restricted AND-parallelism does restrict the parallelism available during program execution because it is likely that many goals will share variables. Another problem with this approach is that it is not always possible before runtime to tell which goals share variables which may mean utilising a runtime check on variables before executing two goals in AND-parallel; this runtime check can be expensive.

2.3 Committed-choice non-deterministic logic languages

The committed-choice non-deterministic (CCND) logic languages have been specially designed to exploit the concept of concurrency so that parallel algorithms can be expressed as logic programs.

The main members of the CCND language group are:

PARLOG Designed and developed at Imperial College, London (see [Gre87]);

Guarded Horn Clauses (GHC) The language chosen for the Japanese Fifth Generation Computer Systems (FGCS) project (see [Uch83] and [Ued86]);

Concurrent Prolog (CP) Designed and developed at the Weizmann Institute, Israel (see [Sha87a]).

2.3.1 Syntax

A CCND program is a finite set of guarded Horn clauses of the form:

$$R(a_1, \ldots, a_k) : -G_1, \ldots, G_m : B_1, \ldots, B_n. \quad (k, m, n \geq 0)$$

where $G_1 \ldots G_m$ and $B_1 \ldots B_n$ are literals.

We use the following terminology to describe the components of a guarded Horn clause:

head for $R(a_1, \ldots, a_k)$;

functor is R;

arity is the number of arguments in the head of R, namely arity k;

guard is formed by the conjunction of guard goals $G_1, \ldots, G_m$;

commit operator is denoted by ":";

body is formed by the conjunction of body goals $B_1, \ldots, B_n$;

clause separator is denoted by ".".

A program query is composed of a conjunction of literals:

$$? - C_1, \ldots, C_n.$$

2.3.2 Semantics

2.3.2.1 Declarative semantics

The declarative semantics of guarded Horn clause programs is similar to that of Horn clause programs. The clause:

$$R(a_1, \ldots, a_k) : -G_1, \ldots, G_m : B_1, \ldots, B_n.$$

is read as:

R is true when $G_1, \ldots, G_m$ are all true and $B_1, \ldots, B_n$ are all true.

2.3.2.2 Operational Semantics

The operational semantics of a CCND program follows a number of phases when a goal is invoked:

head unification The goal is unified with the head of each clause in its relation.

guard execution The guard goals of each of the clauses in the candidate set are executed.

Head unification and guard execution may happen in parallel.

Depending on the outcome of the head unification and guard execution, clauses belong to one of three groups:

candidate clause if both the head unification and guard execution succeed;

non-candidate clause if either the head unification or any one of the guard goals fails;

suspended clause if either the head unification or any of the guard goals suspends but does not yet fail.

If there are clauses in the candidate clause set then one of them is chosen and is **committed** to. If there are no candidate clauses but there are suspended candidate clauses then the calling goal is suspended. If there are no candidate clauses and no suspended candidate clauses then the calling goal is failed, resulting in the failure of the entire computation.
When a clause is committed to, any bindings of variables made in the guard of the clause, which until now have been local to the environment of the clause, are made global. The calling goal is then **reduced** to the body goals in the clause.

Notice that there is no backtracking if a goal fails; evaluation of a CCND program will only result in one solution at most.

2.3.2.3 Variables, suspension and synchronisation

CCND languages have a **single-assignment rule** when dealing with the binding of variables. That is, once a variable has been bound to some value it keeps that value for the duration of the evaluation of the program. Variables cannot become free once bound; there is no Prolog-like backtracking mechanism to enable variables to take on alternative values.

One consequence of this rule is that CCND languages introduce a convention for controlling which goal in a conjunction may bind a variable to a value, and which goals may only inspect the value of the variable. The mechanism introduced to handle this convention is the principal syntactic and semantic difference between the CCND languages.

PARLOG has **mode declarations** to control the way that variables are bound. Each relation has a mode declaration associated with it and the declaration marks each argument as an input argument or as an output argument. For example, the mode

declaration for the append relation is likely to be "`mode append(?,?,^).`" where "?" denotes an input argument position and "^" denotes an output argument position. During head unification and guard execution, any attempt to bind a variable from the environment of the calling goal which is also in an input moded argument position will result in the binding attempt suspending until the variable is bound elsewhere.

In Concurrent Prolog programs the variables in a clause may be annotated as **read-only variables**; they are annotated with a "?" suffix. Read-only variables in a calling goal may not be bound as a result of evaluating that goal. If an attempt is made to bind a read-only variable in a certain clause then that binding attempt will suspend until the variable has been bound by another goal, which in turn results in the suspension of the attempted clause evaluation.

Guarded Horn Clause programs do not have mode annotations. Instead, evaluation of a GHC program follows the rule that no variable from the environment of the calling goal may be bound during head unification and guard evaluation of a clause. Such a variable may only become bound as the result of evaluating a body goal from the clause after commitment. Any attempt, before committing, to bind a variable from the calling environment of the goal will result in that binding attempt being suspended.

2.3.3 CCND languages and parallelism

The operational semantics of the CCND languages has implications for the possible forms of parallel evaluation available:

Restricted OR-parallelism User defined goals may appear in the guard of a clause so that a limited form of OR-parallelism is available when the input unification and guard execution phases of each clause in a relation are executed in parallel. This is limited OR-parallelism, however, because once one of the clause evaluations is committed to the other evaluations are killed off and the OR-parallel search ends.

Stream AND-parallelism A consequence of the single-assignment rule for variables is that AND-parallelism is (intentionally) limited. This rule forces a style of programming called stream-AND parallelism. Given two processes with a shared variable executing in parallel, one process will incrementally instantiate the variable to some data structure, and the other process will consume that structure. The shared variable is like a communications stream on which values are passed between concurrently executing processes.

Figure 2.3 illustrates the stream-AND parallel concept by presenting a program that generates a list of integers and then finds the sum of the list. (The program has been

```
%% PARLOG
mode nat(?,^).
nat(0,[]).
nat(N,[N|T]) :- N>0 : N1 is N-1, nat(N1,T).

mode sum(?,?,^).
sum([], R, R).
sum([H|T], S, R) :- SN is S+H, sum(T,SN,R).

?- nat(100, X), sum(X, 0, Y).

%% Concurrent Prolog
nat(0,[]).
nat(N,[N|T]) :- N>0 : N1 is N-1, nat(N1?,T).

sum([], R, R).
sum([H|T], S, R) :- SN is S+H, sum(T?,SN?,R).

?- nat(100,X), sum(X?,0,Y).

%% Guarded Horn Clauses
nat(0,R) :- R=[].
nat(N,R) :- N>0 : N1 is N-1, nat(N1,T), R=[N|T].

sum([], S, R) :- R=S.
sum([H|T], S, R) :- SN is S+H, sum(T,SN,R).

?- nat(100,X), sum(X,0,Y).
```

Figure 2.3: Program for summing the integer sequence 1...100. It is written three times to illustrate the three main CCND languages: PARLOG, CP, and GHC.

written in PARLOG, CP and GHC to illustrate the main syntactic/semantic differences between them.)

In the query, a process `nat(100,X)` generates a list of natural numbers from 100...1 on the stream `X`; in parallel the process `sum(X,0,Y)` consumes the stream `X`, sums the integers on that stream, and delivers the result of the sum in the variable `Y`. (Note that the semantics of *is/2* are that if the RHS contains unbound variables then its evaluation will suspend until those variables are bound.)

2.3.4 Flatness

The operational semantics of guarded Horn clauses can be made simpler by imposing the restriction of **flatness**. That is, that only system defined tests may appear in the guard of a clause; there must be no user goals. This removes almost all OR-parallel search. Although input unification and guard evaluation for each clause in the relation may still be performed in parallel, since there are now only system goals which consist of tests in the guard, the OR-parallel search is severely restricted.

This has benefits for implementors, especially when implementing for distributed memory machines. OR-parallel search means having separate environments for each branch of the search until commitment. Managing copies of environments across processors is a significant overhead but restricting OR-parallel search removes the need for such environment management.

The main worry with imposing the flatness restriction is whether this severely restricts the evaluation mechanism to such an extent that it becomes impossible to write programs to perform certain types of search problems. This worry has been removed by the discovery that any program with OR-parallel search in the guards of its clauses can be transformed into a program where the search is done in AND-parallel in the body of the clauses. That is, OR-parallel search can be transformed into AND-parallel search[CS87].

Imposing the flatness restriction on the full CCND languages gives flat versions of the languages: Flat PARLOG[FT88], Flat Concurrent Prolog (FCP)[Sha87b], and Flat Guarded Horn Clauses (FGHC)[IMT87].

In this thesis we will work with FGHC. We have chosen it because it is simpler to implement than FCP because it does not have read-only variables. (Flat Parlog and FGHC are essentially the same language.) FGHC is also more representative of the committed-choice languages that are being studied by the majority of other workers for implementation on distributed memory machines.

2.3.5 Other committed-choice logic languages

It is worth quickly mentioning some other committed-choice logic languages that have been developed since the three main languages:

Strand88[FT90] came out of the PARLOG project at Imperial College, London. It is most similar to FGHC, although mode declarations are provided for added efficiency. Unification is restricted to one way assignment of variables in the body of a clause; this simplifies the need to support distributed unification schemes on a distributed memory machine.

Janus[SKL90] is similar to FGHC but restricts clauses to have at most two input variables and one output variable in the head of a clause. This restriction greatly simplifies the evaluation mechanism without greatly affecting the ability to write useful programs.

Fleng[NT88] is similar to FGHC except there are no guards and so no guard evaluation machinery.

2.4 Parallel computers

In this section we will briefly describe the parallel computer architectures that we will refer to later. There are several ways of classifying computer architectures; one of the most popular is the Flynn taxonomy[Fly72]. This taxonomy is based on the organisation of streams: the instruction stream that controls the actions of processing elements; and the data stream, the data that processing elements compute over.

This gives a basic taxonomy of computer architectures:

SISD Single instruction stream, single data stream. This is the conventional sequential processor that has a single processor that receives instructions from a single stream and computes over a single data stream.

SIMD Single instruction stream, multiple data stream. In this group are the vector and array processors. These machines have multiple processors that compute over their own data streams, but they are all executing the same instruction stream.

MISD Multiple instruction stream, single data stream. This suggests that there are many processors performing operations on the same data. To date there are no devices in this group.

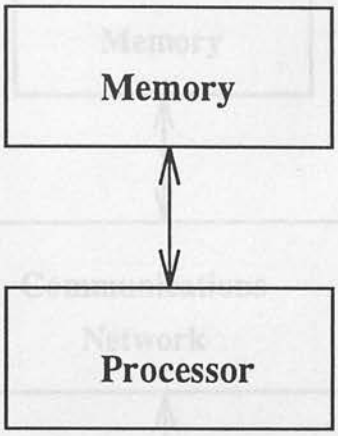

Figure 2.4: Schematic diagram of a conventional sequential computer

MIMD Multiple instruction stream, multiple data stream. The feature of this group is that there are multiple processors computing over their own data stream, and each processor has its own independent instructions stream. That is, each processor may be executing a different program.

There are two main subgroups in the MIMD group: shared-memory machines and distributed memory machines.

Figure 2.4 is a schematic diagram of the SISD (conventional sequential) machine. It shows a single processing element and a single memory element connected via a bidirectional data transfer link.

Figure 2.5 is a schematic diagram of a shared memory parallel computer. It shows that there are multiple processing elements and multiple memory elements that communicate with each other via a communications network. Generally the memory elements form a single physical address space which is the same for each of the processing elements. No processing element controls any particular memory element; all processors have equal access to any memory element. The communications network must resolve any memory conflicts that may arise when two or more processing elements try to access the same memory elements or physical memory address. Processing elements cooperate and communicate with each other by leaving data in certain shared areas: one processor may set an address to a particular value and another processor may inspect that value.

The main advantages to the shared memory parallel machine architecture are as follows:

- memory access time is uniform for each processing element regardless of the memory address accessed;

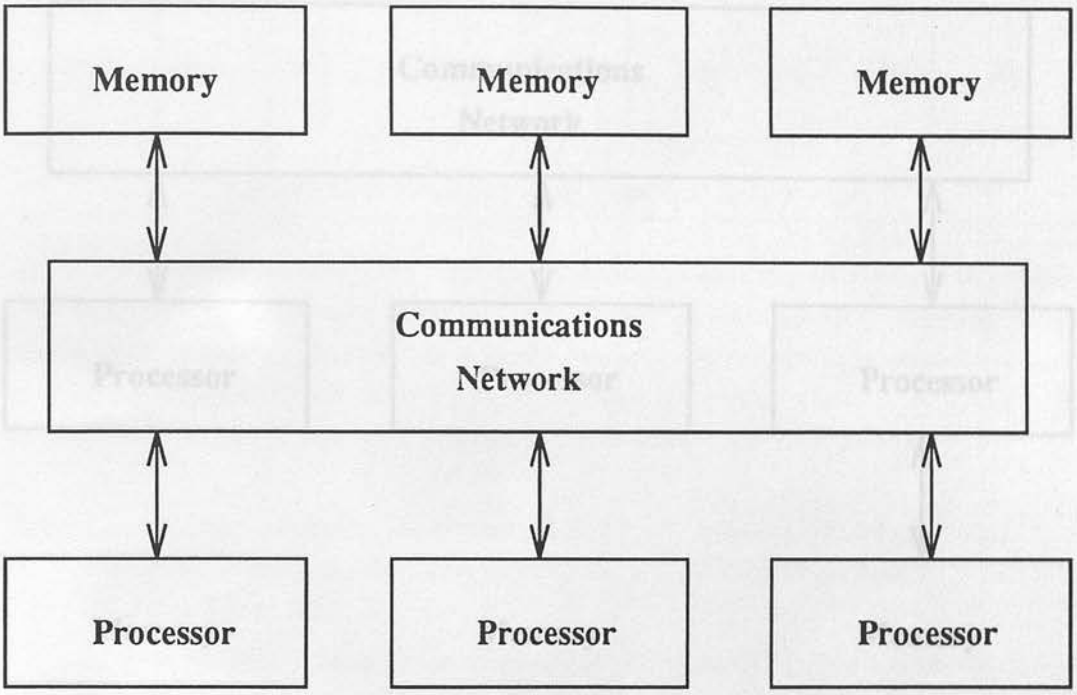

Figure 2.5: Schematic diagram of a shared memory parallel computer

- the shared memory model is similar enough to the sequential machine model to make adapting a sequential program to execute on a shared memory machine possible.

The main disadvantage of shared memory machines is that, as the number of processors is scaled up, the communications network necessarily becomes more complicated and memory access times eventually increase. For this reason there are practical limits on the size of shared memory machines.

Figure 2.6 is a schematic diagram of the class of distributed memory parallel computers. It shows multiple processing elements each connected to a private memory element, and processing-element-memory-element pairs communicate with each other through a communications network. This can also be viewed as multiple sequential machines communicating via a network.

Processing elements do not have equal access to each memory element. They will have fastest access to their private memory element, but to access an alternative memory element requires negotiating with the associated processing element via the communications network. The time taken to access a remote memory element depends to a large extent on the properties of the communications network.

The memory elements do not usually form one physical address space (although some

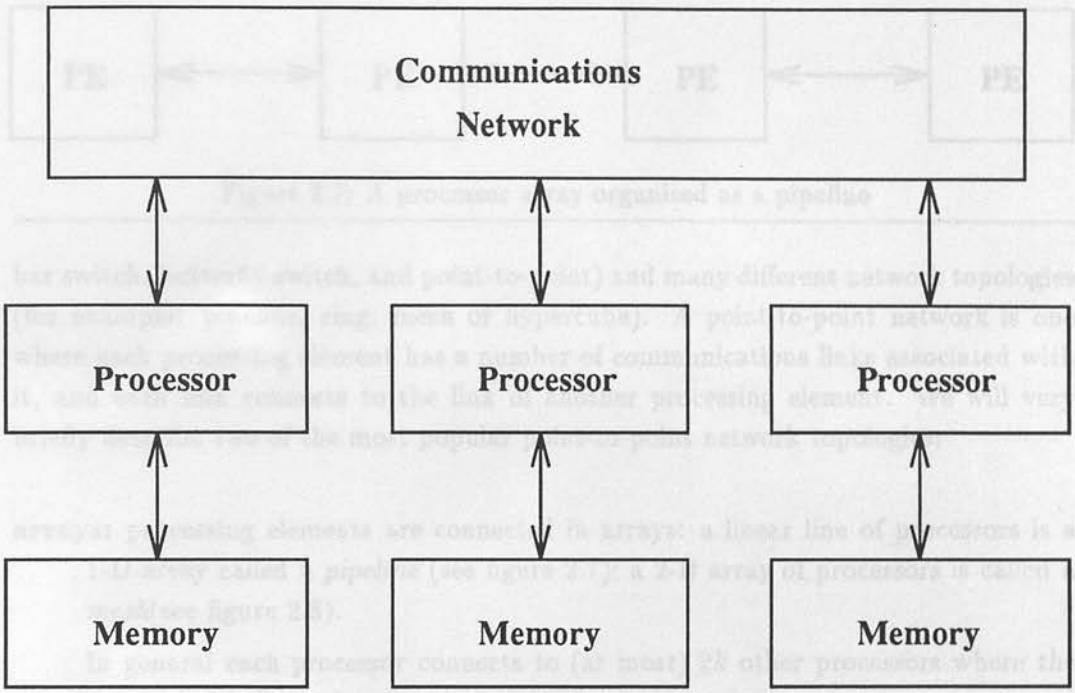

Figure 2.6: Schematic diagram of a distributed memory parallel computer

systems implement a single logical address space) which is why the machines are called distributed memory machines — memory is distributed across the processing elements.

The processing elements cooperate and communicate with each other generally by sending messages to each other over the communications network. Another name for these computers is message passing parallel computers.

The advantages of these machines is that since they do not rely on a shared resource, such as memory, they can support many more processing elements than the shared memory machines; commercial machines with thousands of processors are not uncommon.

The main disadvantage of distributed memory machines, from the point of view of the programmer, is that they are harder to program and obtain speedup from than sequential or shared memory machines. This problem is the starting point of this thesis and is discussed more fully below.

The main physical disadvantage of distributed memory machines is that, as the number of processing elements increases, the maximum time that a message can take between two processors increases. The rate at which this time increases is a function of the network topology and the method by which messages are sent between processors.

There are many different types of communications network (for example: buses, cross-

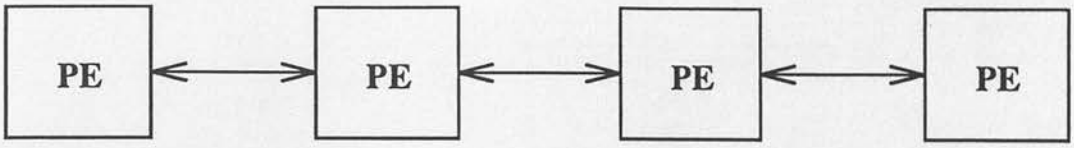

Figure 2.7: A processor array organised as a pipeline

bar switch, butterfly switch, and point-to-point) and many different network topologies (for example: pipeline, ring, mesh or hypercube). A point-to-point network is one where each processing element has a number of communications links associated with it, and each link connects to the link of another processing element. We will very briefly describe two of the most popular point-to-point network topologies:

arrays: processing elements are connected in arrays: a linear line of processors is a 1-D array called a *pipeline* (see figure 2.7); a 2-D array of processors is called a *mesh*(see figure 2.8).

In general each processor connects to (at most) $2k$ other processors where the array is of k-dimensionality.

hypercubes: each processor connects with k other processors, so that there are 2^k processors in the machine; such an arrangement is called a k-dimensional hypercube.

In conjunction with the network topology, the way in which messages are passed between processing elements has a bearing on the efficiency of the machine. Early machines used **store-and-forward** message routing: a message between two distant processors would be stored in intermediate processors and then forwarded to the next intermediate processor until the message reached the intended destination. In this message routing system the maximum delay time of a message depends on how many store-and-forward operations the message goes through, which depends on the number of intermediate processors between the two most remote processors in the network.

The table in figure 2.9 shows some properties of our example network topologies. From this table the hypercube network topology looks very attractive since it has the least fast growing function for the maximum message time. This means that, in a processor with a very large number of processors, the hypercube will deliver messages significantly faster than for the other types of network topology. This is one of the main reasons why hypercube network topologies have been traditionally popular with commercial manufacturers of parallel computers.

An alternative message routing strategy is to check the destination of the message on arrival and route it on-the-fly without storing the message at all. This saves the cost

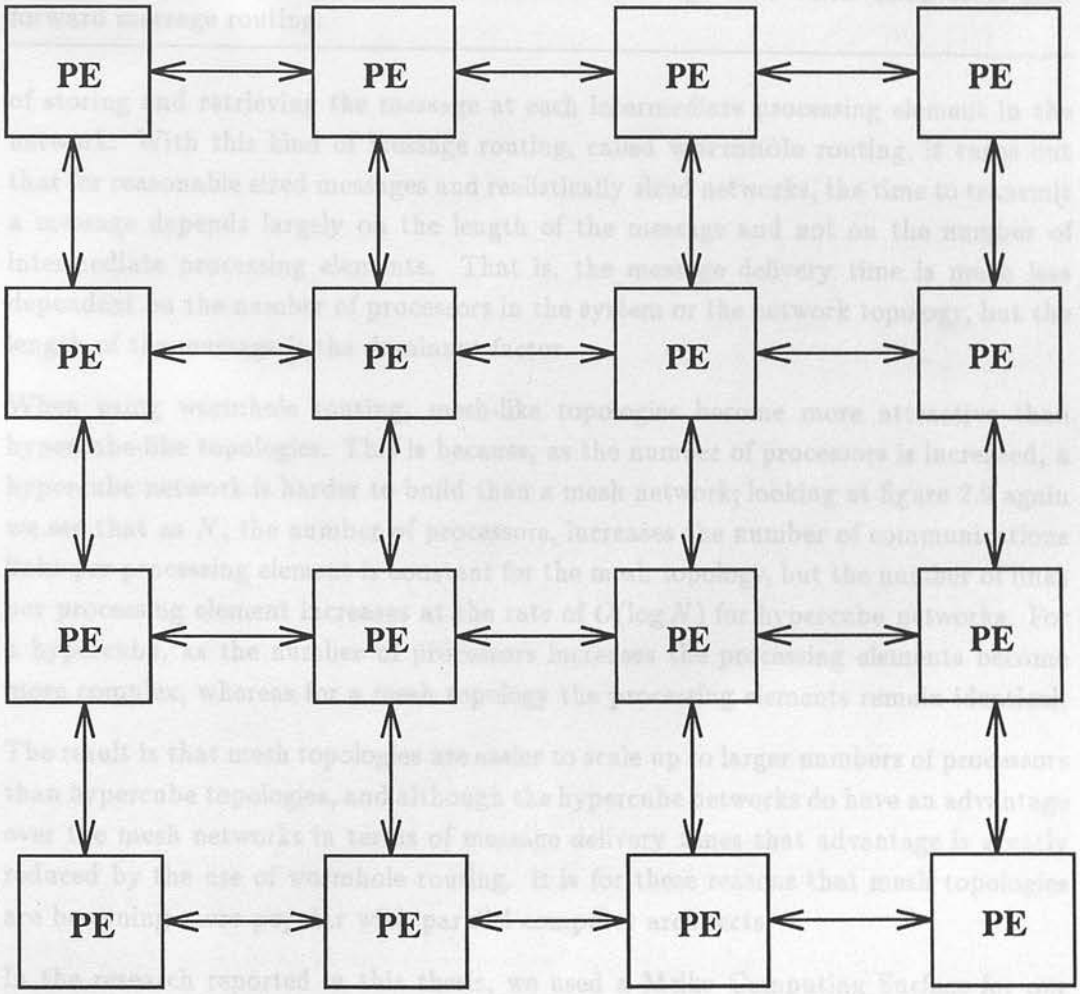

Figure 2.8: A processor array organised as a 2-D mesh

topology	*# of links*	*maximum message time*
pipeline	2	$O(N)$
2-D mesh	4	$O(\sqrt{N})$
hypercube	$O(\log N)$	$O(\log N)$

Figure 2.9: Properties of some network topologies: the number of communication links per processor and the order of the maximum message time when using store-and-forward message routing.

of storing and retrieving the message at each intermediate processing element in the network. With this kind of message routing, called **wormhole** routing, it turns out that for reasonable sized messages and realistically sized networks, the time to transmit a message depends largely on the length of the message and not on the number of intermediate processing elements. That is, the message delivery time is much less dependent on the number of processors in the system or the network topology, but the length of the message is the dominant factor.

When using wormhole routing, mesh-like topologies become more attractive than hypercube-like topologies. This is because, as the number of processors is increased, a hypercube network is harder to build than a mesh network; looking at figure 2.9 again we see that as N, the number of processors, increases the number of communications links per processing element is constant for the mesh topology, but the number of links per processing element increases at the rate of $O(\log N)$ for hypercube networks. For a hypercube, as the number of processors increases the processing elements become more complex, whereas for a mesh topology the processing elements remain identical.

The result is that mesh topologies are easier to scale up to larger numbers of processors than hypercube topologies, and although the hypercube networks do have an advantage over the mesh networks in terms of message delivery times that advantage is greatly reduced by the use of wormhole routing. It is for these reasons that mesh topologies are becoming more popular with parallel computer architects.

In the research reported in this thesis, we used a Meiko Computing Surface for our experiments.

This is a distributed memory machine that uses Inmos T800 Tranpsuters as processors and Meiko's CSTools environment for passing messages between processors, which uses wormhole routing. The Computing Surface has a reconfigurable topology. For the reasons explained above we used it in a 2-D mesh topology. (More details about the organisation of the machine and how we intend to use it are presented in chapter 3.)

2.5 Process distribution

The word process can refer to different things in different contexts. To keep the discussion general, we will view a process as a distinct unit of work that may execute in its own environment (that is, it is likely to have some sort of unique context attached to it).

A parallel program typically consists of a number of processes. Given a suitable computer, the processes may be assigned to processors in such a way that they execute in parallel.

The problem of goal distribution is when and on which processor each process should be executed to obtain the minimum runtime on a given computer system.

In general, the processes constituting the program have a complex relationship in that there may be **dependencies** between processes. Dependencies occur when processes need data from other processes before they can commence, continue or complete execution. Given two processes, A and B, we can define different types of relationship between them:

Fully independent A and B can execute in parallel without exchanging data.

Fully dependent A needs data produced by B before it can commence execution. In this case, execution of A is fully dependent on the execution of B.

Mutually dependent A needs data produced by B to continue execution, and at the same time B needs data produced by A to continue execution. A and B are said to **coroutine.**

Process dependencies place time ordering constraints between processes. Processes will also execute in differing amounts of time and communicate values at different frequencies and sizes.

The problem is how to distribute processes over a distributed memory machine to minimise runtime. General ways to minimise runtime are to:

- maximise the number of processes that can execute in parallel;
- minimise the amount of communication between processes;
- minimise the distribution of processes between processors.

Notice that these hints are to some extent conflicting. It would be simple to reduce the amount of communication between processes to zero by executing the program on

a single processor, but that would also minimise the number of processes that execute in parallel.

Goal distribution methods can be placed broadly in a two-dimensional space. The first dimension relates to *when* goal distribution is decided. This ranges from **static** to **dynamic**; that is, goal distribution can be decided before runtime or during runtime. Dynamic goal distribution strategies can be placed along a range according to *how* the processes are distributed. This ranges from **demand driven**, where idle processors ask other processors for spare processes, to **offloading**, where processors with excess work send spare processes to another processor.

The second dimension relates to *what* decides goal distribution. This ranges from **manual** to **automatic**, where goal distribution is decided by the programmer or program analysis respectively.

2.6 Goal distribution and the committed-choice logic languages

Conventional concurrent language implementations for distributed memory computers, such as FORTRAN and C with message passing extensions, tend to have several common features:

Large grains: Processes tend to involve a substantial amount of processing compared to the amount of communication they make. That is, they have large **granularity**.

Static process structures: The type and number of processes are statically defined; that is limited or no process creation at runtime.

No process migration: Processes tend not to move between processors. Process allocation is decided statically.

Fixed communications: Communications channels between processes tend to be point-to-point and are defined statically.

In addition, communications channels tend not to be treated as first class objects; that is, communications channels cannot be communicated over communications.

These features make goal distribution easier in that the problem becomes how to map a statically defined set of processes, with a statically defined set of communications channels between processes, onto a fixed computer architecture, although it is still a difficult problem.

In contrast the features of committed-choice logic languages are:

Fine grains: Processes tend to be very fine grained in that a reduction of a goal typically involves little work.

Dynamic process creation: Goals[1] reduce to subgoals dynamically as the program executes.

Dynamic communications: Since goals communicate through shared variables, arbitrary communications links between goals can be created.

Also, shared variables may contain lists of other shared variables; that is, communications streams may contain other communications streams giving more scope for complex dynamically changing communications structures between goals.

2.7 Current Approaches

Here we look at current approaches in trying to solve the goal distribution problem for the committed-choice logic languages. We describe each of the main approaches. Discussion of the merits of each approach are postponed until after the end of the chapter.

2.7.1 The Japanese FGCS Project

Researchers on the Japanese Fifth Generation Computer Systems Project at ICOT have designed and built many sequential and parallel implementations of various concurrent logic languages over the past ten years. Currently their attentions are focused on their own concurrent logic language Guarded Horn Clauses[Ued86] which belongs to the committed choice family of languages. From the outset of the project they decided that the basic language should not include primitives for any form of control, including goal distribution. Their argument is that the control can either be derived by analysis or that an orthogonal control language could be developed to allow programmers to control program execution for a given architecture.

ICOT researchers have developed the KL1 language[KC87], a kernel language based on Flat GHC. KL1 has meta-programming annotations for the description of goal management. There are three main types of annotations:

[1] We will use process and goal synonymously throughout the rest of the thesis.

Shoen which is used to group goals as a unit and provide communication streams to control the behaviour of the unit;

Priority pragma which is used to give a goal inside a shoen an execution priority value;

Goal pragma which is used to indicate on which processor a goal should be executed.

This approach is static — pragmas map processes to processors — and manual — the user supplies the annotations.

2.7.2 Hidaka et al.

Hidaka et al., at the University of Tokyo, have developed an automatic analysis tool for committed choice logic languages[HKTT91]. Given a program, there are points in the program where:

- memory is allocated for new data;
- goals are created.

These are termed **load partitioning points.** Each new datum or new goal will reside on a processor. The problem is: *Which processors should they be assigned to to give the minimum runtime?* For each load partitioning point a **load partitioning tactic** will be selected. The authors recognise four such tactics:

A Select the processor with the lowest load[2];

B Select the processor where the parent goal resides;

C Select the processor on which a particular datum resides;

D Select the same processor as selected by using tactic A at a different load partitioning point of memory allocation in the same clause.

The load partitioning strategy for a particular program is expressed in the form of adding load partitioning tactic annotations at load partitioning points in the program. The problem now becomes: *Given a program, how to generate a new program which is the old one but with the partitioning points annotated with tactics which when the new program is run will give near optimal speedup?*

[2]They assume a computer architecture that has special hardware for finding the processor with the lowest load, which is not the case for most general purpose machines.

This is done by a profiler. The profiler has a complex execution which we will not describe in detail here. In essence, it simulates execution of the problem program on a computer which has: an infinite number of processors; constant time between successive generations of goals; two types of memory access latency: local and remote. At the same time as simulating program execution the profiler builds up a large data structure recording information about **objects**; that is, goals and data. The information is comprised of sets of conditions of the form: if we choose these tactics at these partitioning points then this object will reside on such and such a processor. In other words, the profiler performs an all solutions search of the implications of choosing each tactic at each partitioning point[3].

During execution the profiler also collects information on:

- the number of local memory references made to a datum assuming certain tactics at partition points;
- the parallel degradation caused when assuming certain tactics on allocating goals to processors at partition points.

It is claimed that using the output of the profiler an assigning of tactics for partitioning points can be done using a simple non-heuristic algorithm giving a program with near optimal speedup. The essence of the algorithm is that:

1) From the profiler information, choose a partitioning point that has a tactic that if chosen will result in the largest number of local memory references. If parallelism degradation might result from choosing that tactic then choose the next best one.
2) If no such tactics exist then select tactic A for that point. Propagate all values through the information base.
3) Repeat 1) to 2) until no more tactics can be chosen for partitioning points on the basis of memory references.
4) For the remaining partitioning points choose tactic B unless parallelism degradation will result in which case choose tactic A.

Having done this profiling and computation of the information base, the result is a program with tactic annotations for partitioning points which if run should give near optimal speedup given the assumptions made about the simulated computer. The profiler can be run in an evaluation mode to gather performance statistics of a simulated

[3]This is the essence of the profiler although an oversimplification.

run of the annotated program. Such profiles can then be compared against profiles of programs annotated in other ways.

The first thing to strike us about this approach to static goal distribution is the time it takes to do the analysis. For a program to calculate the first 100 prime numbers the time taken in analysis on a Sun4/260 is 235.4 seconds (4 minutes). The same result for the 6-queens program is 3875 seconds (about an hour). To counter this, the authors claim a reduction by a factor of 2 in remote memory references with the optimal partitioning as opposed to a simple partitioning, while at the same time parallelism has reduced by around a third. Being optimistic, using this partitioning system will give an increased speedup up of 2 over simple methods while incurring a large analysis cost. The analysis cost will grow with the size of the computation which means that it will become prohibitive for programs of even a modest size to perform the analysis. A good point of this approach is that it is the only one so far to demonstrate a framework for an automatic, mostly static goal distribution scheme.

2.7.3 Foster

In [Fos90], Foster approaches the problem by developing a set of tools to aid the programmer in partitioning and scheduling goals for a particular computer architecture. His thesis is that partitioning and distributing goals to processors should not be done by the underlying operating system since this is bound to be inflexible and machine dependent.

Instead it is proposed that schedulers should be written in the source language, in this case Strand-88, as metaprograms which will manipulate the user program as data. In the example given in the paper, the scheduler code consists of the definition of a manager process and a worker process. At runtime the manager will pass goals[4] to workers. Workers receive and reduce goals before returning the resultant (sub-)goals to the manager for redistribution. Workers will only return goals that can be executed immediately; that is, goals that have ground input arguments. This removes the cost of allocating processes that will suspend immediately.

An interface language is used to interface between schedulers and user programs. Statements in this language define:

- the initial goal called in the program;
- a set of task declarations.

[4] They are called tasks in the paper.

A task declaration declares:

- that a certain goal in a particular clause in the program is a task.
- the tasks on which this task is dependent; that is, the tasks that must have terminated before this task can commence execution.

These declarations are made by the programmer.

The program interface declarations are then composed with the user program using a source-to-source compiling algorithm to produce a program that will compute scheduler protocols. At the same time as compiling in the scheduler interface, statistics gathering relations may also be compiled in. The result is a self-scheduling program that can be executed in conjunction with any one of the schedulers, chosen from a predefined library of schedulers, and which will also dump statistics about itself during execution.

Given a graphical tool to analyse the execution dump the user can see how well the program executes with the given scheduler. The user can choose another scheduler to see how the execution compares with the first. By trial-and-error, and possibly some insight, the user should be able to find an adequate scheduler.

Currently, the authors point out that this approach works only for programs for which a **directed partition** can be specified; that is, programs with acyclic task dependencies.

2.7.4 Tick

Tick[Tic90] proposes a simple compile-time method for analysing the **weight** of computation of a procedure. The idea behind the analysis method is:

> "a procedure's granularity is the sum of the granularities of the procedures it calls, and the granularity of a self-recursive clause is 1 by definition."[5]

The weights are then associated with corresponding goals in an execution of the program. A scheduler can then make scheduling decisions by comparing the weights of goals in the run queue.

The analysis is done by constructing a weighted directed graph. There is a node for each predicate in the program. For every call to a predicate p_b in the body of procedure p there is an arc in the graph from node p to node p_b. Weights for each node, and hence predicate, are calculated as follows:

[5] [Tic90]

- Tail-recursive procedures are assigned a weight of one, except for a few that are explicitly declared by the user as **perpetual** which are assigned the weight of 0.

- The weight of a non-tail-recursive node is calculated as the average of the weights of each clause making up the procedure. The weights of a clause are calculated by summing the weights of the procedures called in the body of the clause.

As might be expected, this method has problems when considering mutually recursive procedures. Consider two mutually recursive procedures a and b. Tick solves the problem of finding $weight(a)$ and $weight(b)$ by arbitrarily choosing say node a and **marking** it as used. When calculating the weight of a, the weight of b needs to be calculated (since a calls b) which in turn is calculated from the weight of a (since b calls a). Since a has been marked it is given the value of 1 for the purposes of calculating the weight of b. This assignment of 1 to a marked procedure cuts the mutual recursion albeit in an arbitrary way. This cutting heuristic does result in some unusual weight assignments in programs with mutually recursive procedures but Tick claims that this effect is not significant in practice.

Each goal instance in an execution of the program now has an associated weight which can be read from the appropriate predicate node in the graph. Each processor of a parallel computer has a goal queue organised by the processor's scheduler. The scheduler keeps the goal queue ordered into goals of ascending weight so that goals with low weight are reduced first.

Goals with high weight are *stolen* first by an idle processor which is done simply by removing the first goal at the appropriate end of the queue. In this way, goals with high weight will be distributed out to other processors, but those with low weight will be executed on the local processor.

This analysis method has been implemented and tried on a parallel implementation of FGHC executing on a Sequent Symmetry shared-memory parallel machine. Tick reports disappointing speedups of less than 10% when using weighted goal ordering against using goals ordered by their appearance in the source code. He attributes this low speedup figure to the low cost of spawning processors on the machine, which means that the weight of goals is not significant, and the nature of the test programs tried.

This approach is not a full goal distribution strategy but really a goal ordering strategy which can be used as a component of a goal distribution strategy. Although Tick's results are disappointing for his shared memory machine implementation it would be interesting to see how the same strategy fares in a distributed memory environment where the cost of spawning a goal on a remote processor (migrating a goal) is likely to be more expensive than on a shared memory machine.

We propose this goal ordering strategy in some of our goal distribution strategies in order to evaluate its effectiveness on distributed memory machines.

2.7.5 Crammond

Crammond[Cra90] describes three similar dynamic scheduling algorithms intended for use with committed choice logic languages on shared-memory architecture parallel machines.

In the first algorithm, each processor has a **local work pool** of processes. A processor removes work and adds work to the *front* of its local work pool. This means that a processor schedules work locally in a **depth-first** manner. When a processor exhausts work from its own local work pool it explores the work pools of other processors until it finds one with work. It then **steals** a process from the back of the pool. This means that work is distributed across processors in a **breadth-first** fashion. This makes sense in a shared-memory environment since the steal operation does not disturb busy processors.

The second algorithm extends the first by tagging processes as either **private** or **public**. Only public processes may be stolen by other processors. A processor periodically turns processes from private into public. This puts a slight overhead on busy processors.

The third algorithm has completely private local process pools and a single global public process pool. This has the advantage that idle processors need only look in one place to find processes. Again this causes overhead for busy processors because they have to move processes from their private pool to the global one.

All three algorithms were implemented for a Parlog system running on Sequent Symmetry and a BBN Butterfly, both shared memory computer architectures. After completing a set of benchmarks Crammond concluded that the second algorithm is probably the best of the three for shared memory computers running this sort of system.

The results are likely to be different for distributed memory machines since they introduce non-uniform memory access times between processors. It would be interesting to know how these algorithms would perform on distributed memory architectures. We evaluate some of Crammond's goal distribution schemes when used on systems executing on a distributed memory machine.

This approach is dynamic, demand driven and fully automatic.

2.7.6 Shapiro et al.

The designers of Concurrent Prolog argue that instead of automatic goal distribution strategies, language designers should provide a simple set of constructs to support mapping[TARS87]. They describe a four layer methodology for executing a program on a physical machine:

Application layer Goals in the body of a clause may be annotated with **turtle mapping annotations**. The annotations describe a relative path from the current processor to the processor on which the goal should be executed. The user must provide the program annotations.

Transformation layer This layer transforms the annotated goals into messages which are passed to the underlying virtual machine. The transformation layer can be described by a meta-interpreter which either reduces local goals or puts a message on an appropriate output stream to execute an annotated goal on another processor.

Virtual machine layer This layer describes a monitor that executes on a processor in the virtual machine. It is written as a CP program that receives messages to execute goals on input streams, and send messages to execute remote goals on output streams.

When mapped to a physical machine, a monitor is executed on each physical processor. Adjacent processors' monitors share a variable for each message stream so that goals may be passed between processors.

Physical machine layer This is the physical machine itself.

The aspect of this system that most interests us is the annotations at the application layer since this describes the basic mechanism by which processes are mapped to the machine (if we assume that the virtual machine is of the same topology as the physical machine). A number of Concurrent Prolog programs with annotations are described in a paper by Shapiro on systolic programming[Sha87c].

As an example of the annotations used we show an example program from that paper, a program to compute prime numbers using the sieve of Eratosthenes method (see figure 2.10). It is intended that the program be executed on a collection of processors organised in a ring topology: hence the `G@forward` annotation which causes the goal "`G`" to be executed on the next processor forward from the current one. Executing this program would result in the *integers/2* goal executing on the first processor but the *sift/2* goal would be sent forward to the next processor. On this new processor the sifter would deposit a *filter/3* goal, which filters multiples of 2, and would then

```
primes(Ps) :-
  integers(2,Is), sift(Is?,Ps)@forward.

integers(N,[N|Is?]) :-
  N1 := N+1, integers(N1?,Is).

sift([Prime|In],[Prime|Out1?]) :-
  filter(In?,Prime,Out),
  sift(Out?,Out1)@forward.

filter([N|In],Prime,Out) :-
  0=:=N mod Prime : filter(In?,Prime,Out).
filter([N|In],Prime,[N|Out?]) :-
  0=\=N mod Prime : filter(In?,Prime,Out).
```

Figure 2.10: Sieve of Eratosthenes

send itself on to the next processor. On that processor it would deposit a process to filter out multiples of 3 and then send itself to the next processor where it deposits a process to filter multiples of 5 .... So the first processor generates integers, the second processor filters out multiples of 2, the next multiples of 3, the next multiples of 5, and so on.

Shapiro notes that this is not a very efficient mapping because a feature of this particular program is that the streams between the filter processes will have lower numbers on them the further they are from the integer process. Instead of the linear mapping above it might be better to try an exponential mapping where the number of filter processes deposited on each processor rises exponentially as the computation progresses. To do this one can substitute an expression instead of just the plain `@forward` annotation:

```
sift([Prime|In],[Prime|Out1?]) :-
  filter(In?,Prime,Out),
  sift(Out?,Out1)@forward(2/Prime).
```

where `@forward(2/Prime)` means spawn forward 2/Prime times.

These annotations can be very elaborate:

```
@(right, forward(2^(N-1)),left,forward(2^N),left)
```

It would seem that these turtle annotations are good for programs that generate regular process structures, for instance pipelines or trees of processes. It is hard to imagine

how programs that create highly irregular process structures or process structures dependent on input data would be mapped using turtle annotations.

This method also makes the simplifying assumption that the virtual machine has an infinite number of processors. For example, what would be the effect of executing the *primes/1* program on a computer with only four processors?

2.8 Discussion

It would seem that the idea of dynamic process distribution strategies for the CCND languages has been mostly neglected.

Shapiro's group has rejected the idea believing that distributing processes to processors is best left to the user and that as more is understood of parallel algorithms the easier this will become for users. To support this approach they have decided to offer annotations for process mapping. These annotations can become very elaborate which suggests that the user must have a very intimate knowledge of the behaviour of the algorithm and of the virtual machine, if not also the physical machine, on which the program will execute. It is also very hard to see how this approach could be applied to programs that would need a non-regular mapping.

A similar approach has been followed by the Japanese FGCS group. They provide a control language for control of goal execution and distribution. Again the control instructions are supplied by the user which suggests that the user must have an intimate knowledge of behaviour of the program and the system on which the program will execute. It is also likely with this approach that the control instructions will need to be modified if the program is to execute on another type of computer.

The group at Tokyo University have designed a system which basically simulates most of the possible execution strategies for a program on the target machine simultaneously. The result of the simulation is then analysed to find the best way of placing goals and variables to minimize execution time. The program is then annotated to give the desired behaviour at runtime. The biggest problem with their system is the time that it takes to run the simulation; it takes an hour to simulate a 6-queens program, a program that will run in a few seconds on a sequential machine. It is clear that this is impractical for programs that need frequent reconfiguring or that will only be executed a few times in their lifetime. An alternative is to perform the simulation for a smaller problem and then hope that the annotations derived are good for the full problem execution. Such scaling up has yet to be shown to work.

The work done by Foster requires the user to add process dependency declarations to the program. The resulting program can then be automatically transformed into

a program that will be executable with a scheduling program taken from a library of schedulers. Foster claims that even very large programs need only a few declarations to make the system useful (less than 1%). This approach does not describe any process distribution strategy in itself although the system provider or user would need to write the programs in the scheduler library. The major limitation of this system is that it only works for programs for which a directed acyclic call graph can be found: although such a graph can be found for the majority of programs it restricts the form of parallelism allowed to restricted AND-parallelism. Stream AND-parallelism is completely excluded; any program consisting of, for example, a pipeline of producer-consumer processes will execute sequentially. This approach obviously suits programs that have a large component of restricted AND-parallelism, for instance programs using the divide-and-conquer paradigm, but will not suit the majority of programs written in committed-choice languages which tend to rely heavily on stream AND-parallelism.

The work done by Crammond describes fully automatic execution strategies for the CCND languages. There is no burden on the user to supply annotations to gain full benefit of the system. His approach is to have a goal queue for each processor. One end of the queue is used by the processor in a stack-like fashion for removing and adding goals that may be executed locally. The other end of the queue can be accessed by other processors who may steal work when their own goal queues are exhausted[6]. The main point to mention about Crammond's work is that the goal distribution strategies are designed to execute on a shared memory computer. It is not clear whether the same strategies used in a distributed memory environment would still be valid. In particular, in a shared memory implementation, the time taken to access another processor's goal queue or to transfer goals between process queues does not play a role, since these will be constant time operations. These issues are likely to be considerations for an implementation on a distributed memory machine.

The work done by Tick can be seen as a refinement of Crammond's ideas. Crammond orders local goal queues from the program context. That is, goals are added to the goal queue in the same order as they appear in the user program: the leftmost goal will be executed first and the rightmost goal executed last. Tick proposes a method to order goals by a notion of weight and describes an algorithm for heuristicly calculating the weight of each predicate in the program. The idea is that the goal queue can be ordered by weight of goal with the goals with highest weight sent out to idle processors first. If processors are kept as busy as possible with the highest weighted goals then they will become idle less often and so avoid the time consuming business of stealing spare work from other processors. This approach has been added to a shared memory implementation but Tick reports a speedup improvement of only approximately 10%. He suggests that this disappointing result is due to the fact that it makes very little

[6]We have adopted this approach in our work.

difference in a shared memory implementation because stealing work from another process is not very time consuming. The algorithm may have better success in a distributed memory implementation.

2.9 Summary

This chapter consists of two parts: preliminary concepts needed to understand the thesis and a literature survey.

In the preliminary concepts part:

- We have described the main sources of parallelism in parallel Prolog implementations — OR-parallelism, AND-parallelism and restricted AND-parallelism — and that these are not easy to take advantage of on distributed memory machines.
- We have presented the declarative and operational semantics of guarded Horn clauses, the logical basis of the committed choice non-deterministic (CCND) languages.
- We have described the principal committed-choice languages — Concurrent Prolog, Guarded Horn Clauses, and PARLOG — the main differences between them and their flat variants.
- We have described the main sources of parallelism in the committed-choice langauges — restricted OR-parallelism and stream AND-parallelism.
- We have briefly discussed the advantages and disadvantages of shared memory and distributed memory computer architectures, and we have discussed two forms of message routing strategy for the distributed memory computers — store-and-forward and wormhole routing.

In the literature survey part:

- We have described the major approaches taken by researchers in providing goal distribution mechanisms for implementations of the committed-choice languages.
- And finally we discuss the merits and demerits of each approach. We find the approaches taken by the Shapiro group and the Japanese FGCS group, that of annotating programs, to be too burdensome for the user.

  Foster's approach restricts the execution of programs to restricted AND-parallelism which excludes a large set of programs that are essentially stream AND-parallel.

The group at Tokyo University analyse a program through abstract simulation and then provide annotations for an optimum execution of the program. The problem with this approach is that the simulation is very time consuming.

Crammond's approach is probably the most promising; it is a fully automatic demand driven dynamic approach to process distribution. It has yet to be tried on a distributed memory computer.

The work done by Tick can be seen as a refinement to the algorithms used by Crammond. Tick's results with shared memory implementations have been disappointing but this does leave open the possibility of better results with a distributed memory computer implementation.

Chapter 3

Framework

3.1 Introduction

Here we will briefly state the framework in which our experiments with automatic dynamic goal distribution strategies for the CCND languages were carried out. Each part is described in more detail in subsequent sections of the chapter.

We did the following:

- implemented an interpreter for the CCND language Flat Guarded Horn Clauses. The interpreter is designed to execute on a Meiko T800 Transputer array configured as a 2-D mesh topology;
- implemented goal distribution strategies to handle the goals in the system.

There are two dimensions to these strategies:

Demand driven: We used demand driven strategies: an idle processor asks other processors for spare goals. There were two strategies in this dimension: the *nearest-neighbour* strategy where an idle processor asks adjacent processors in the mesh for spare goals; and *all-processors* where an idle processor asks any processor in the system for spare work.

Weights: We implemented two ways of ordering goals for execution and distribution: *weighted* where Tick weights are used to order goals; and *non-weighted* where goals are ordered by the order in which they occur in the source program.

This gives four different goal distribution strategies:

AP-NW all-processors — non-weighted;

AP-W all processors — weighted goals;

NN-NW nearest-neighbours — non-weighted;

NN-W nearest-neighbours — weighted;

The system executed a suite of benchmark programs and the following parameters were varied:

- the number of processors in the system;
- the scheduling strategy;
- the test program.

The results of each run of the system were some output statistics which were analysed to generate performance measures. The goal distribution strategies can be discussed and compared through these measures.

Each part of this framework is now described in more detail.

3.2 Processing agent model

Here we show our model with which to describe distributed CCND language implementions.

A distributed memory machine is a set of processing elements that are each comprised of the CPU, some memory and some communications links. The communications links link the processing elements together in some topology.

Similarly, our distributed language implementation consists of a processing agent on each processing element. The agent manipulates the local memory and communicates with other agents by manipulating the communications links. Each agent is identical on each processor although each has its own unique identifier.

The agent has several subagents three of which are:

reduction agent takes a goal and reduces it to subgoals;

communications agent handles incoming and outgoing messages;

scheduling agent decides on the order in which goals are reduced.

3.2.1 Reduction agent

Given a goal, the reduction agent attempts to reduce it to a set of subgoals. This will be described in the next chapter when we present our FGHC interpreter implementation.

3.2.2 Communications agent

The communications agent handles incoming and outgoing messages, and relays their information to the appropriate subsystems. We will not examine in detail how the communications agent might operate but instead describe the messages it needs to handle.

There are basically three types of message:

- goal messages;
- binding messages;
- system messages.

Goal messages are about the management of goals, like sending out goals, and sending out requests for goals. There are at least three messages to handle goals:

goal_tell(T,F,G) is for sending the goal G from the agent F to the remote agent with identifier T.

goal_ask(T,F) is sent to the agent with identifier T and is asking for a spare goal from the agent on that processor. It is from the agent F.

goal_ask_nak(T,F) is sent to the agent T telling it that the agent F has no spare goals.

Where bindings are concerned we consider variable bindings to be distributed across the machine; an agent will have its local bindings in its local memory. To have access to the variable bindings of a remote agent we need to introduce at least two messages:

binding_tell(T,F,V,X) which is sent to agent T from agent F telling it to bind variable V to the value X.

binding_ask(T,F,V) which is sent to agent T by agent F asking for the value of variable V.

The last set of messages are system control messages: messages to start, stop, and abort the system. Typically messages in this class might be:

task_start allowing the agents to start reducing goals.

task_term to tell an agent that the computation has terminated.

task_abort to tell an agent to terminate the computation early because of some system error, like lack of memory.

3.2.3 Scheduling agent

The scheduling agent controls the local goals and the order in which they are reduced by local or remote reduction agents. We now present a model of the scheduling agent.

The goals are scheduled on a **runnable goal queue** or **run queue** for short. The run queue has a *local end* and *remote end.* Goals are removed from the local end for execution by the local reduction agent; goals are removed from the remote end for execution by a remote agent.

The scheduling agent accesses the run queue through an interface of four operations:

local_schedule(P) pushes a goal onto the local end of the goal queue P on the goal queue.

local_deschedule()→P pops a goal from the local end of the goal queue and returns it as P.

remote_schedule() initiates steps to provide a goal for execution by a remote agent.

remote_deschedule() initiates steps to remove a goal from a remote goal queue for execution by this agent.

3.3 Selecting strategies

In this section we describe the ideals to which we would like our goal distribution strategies to conform, and we then describe the strategies that we have chosen to explore in this thesis.

3.3.1 Ideals

We will outline our ideals for a goal distribution strategy for the CCND languages on distributed memory machines:

Minimum execution time: This is our main reason for executing programs on a parallel processor.

Fully distributed: The strategy should be operated by every processor in the computer. This eliminates communications bottlenecks due to a large number of worker processors trying to communicate with a small number of master processors. With a fully distributed strategy all processors are masters and workers.

Scalable The strategy should be applicable whether the computer has just 2 processors or has 2000 processors. This means that the strategy should not use resources that will not scale up to large numbers of processors. An example might be keeping a 1 MByte cache of data last sent from each processor; this might be acceptable for 2 processors but would be a problem for even a 10 processor machine.

Low overhead Any goal distribution strategy will incur overheads, in time and memory, due to collecting its own information about how and when to distribute goals. These overheads should be kept to a minimum.

Full parallelism The strategy should be able to support the parallelism of the language and not impose any restrictions.

Zero user burden There should be as little involvement from the user as possible. Users would like to be able to write the program and execute it without having to concern themselves with such things as hardware topology or efficiency issues.

In view of the fact that little work has been done on goal distribution for the CCND languages, we can add another ideal to the list above: **simplicity**; we will start with the simplest possible strategies first.

Our aim is to keep to these ideals as closely as possible. If relaxing one or more of them is likely to give an advantage then we will do so. For example, if the user has to make some effort in program annotation that may decrease execution time, and that effort is reasonable, then the possibility would be considered.

3.4 Choosing strategies

We chose to investigate a combination of distribution strategies inspired by the work done by Crammond and Tick described in the previous chapter.

In the following sections the strategies are summarised and the reasons for choosing them are given.

3.4.1 Crammond

The goal execution strategies of Crammond described in the previous chapter would fit most of our ideals if we were designing for a shared memory machine.

The algorithms are distributed in that each processor controls its own queue of goals and decides how to manage that queue in the same way.

Each processor has a very simple task when deciding which goals to execute:

- if there are goals on the local queue then remove the first goal from the front of the queue and try to reduce it;
- if there are no goals on the local queue then steal work from the back of the queue of another processor;
- new work generated is added to the front of the local queue to be processed in the order in which it appears in the source program.

In other words, with respect to the source program, a processor executes goals from its goal queue in a depth-first left-to-right manner and distributes goals to other processors in a breadth-first manner.

We can see how this arrangement can be easily redesigned for a distributed memory parallel machine. Instead of stealing work from another processor by accessing its goal queue in shared memory, an idle processor would send a message to another processor asking for spare work.

One question we might ask ourselves is why specifically use a demand driven strategy? Instead of idle processors asking for work, why not have processors with too much work offload spare work onto other processors?

There are a number of reasons why we found that approach unattractive:

- Offloading causes busy processors to be interrupted to see if they need to distribute spare work to other processors. The demand driven approach places the

burden of finding work on processors that are already idle and doing nothing. It is better to leave busy processors uninterrupted for as long as they have work.

In a scenario where all processors have plenty of work this means that they can execute goals uninterrupted. In the offloading scheme, even though all processors are busy they may try to offload work onto other already busy processors.

- It is difficult to decided the threshold that defines when there are spare goals that should be sent out. If the threshold is set too low then processors may spend a lot of time sending out goals to other processors that do not need the work. If the threshold is set too high then some idle processors may not get enough work.

  In contrast, in a demand driven system it is simple to calculate when a processor has no work — when it has no goals to execute.

We would therefore argue that the demand driven strategies satisfy our simplicity ideals better than do offloading strategies.

3.4.2 Tick

Tick suggests that ordering the goal queue by weight may give better performance than ordering goals as they appear in the source code. The idea behind this strategy is that sending out goals to remote processors may be costly and that, to minimize costs, it is good to send out goals that will keep the idle processor busy for the longest time.

Not only that, there is a cost associated with spawning a goal on a remote processor: the cost of sending the goals and its arguments to the remote processor so that it can be evaluated. If this cost is greater than the time taken to execute the goal locally, then a net loss of performance results. It therefore makes sense to send out large goals to remote processors to minimize this performance loss, and to execute small goals locally.

The result should be that all processors are busily executing large goals, keeping idleness as low as possible, and so maximising parallel execution and minimising messages asking for goals.

To this end Tick proposed a method of calculating the size of atoms (goals) in the source program. The method is quite simple:

- system goals have a weight of 0;
- tail recursive predicates have a weight of 1;
- any other predicate has a weight of the sum of the weights of the predicates divided by the number of clauses attached to the predicate;

- the user can annotate certain predicates to be perpetual; these have a weight of zero.

There are rules to deal with mutually recursive predicates to cut the recursion on weights.

Using this method it is simple to analyse programs and give predicates weights, so the method is virtually automatic; the only user input is to annotate certain predicates as perpetual but this does not seem to be too much of a burden.

```
append([], L, R) :- R=L.
append([H|T], L, R) :- R=[H|R1], append(T, L, R1).

nrev([], R) :- R=[].
nrev([H|T], R) :- nrev(T, NT), append(NT, [H], R).
```

Figure 3.1: FGHC program for naive reverse

As an example consider calculating the weights for the naive reverse program (see figure 3.1). The first clause of both *append/3* and *nrev/2* are essentially unit clauses and are discounted. The second clause of *append/3* is a simple recursive clause (*append/3* reduces to *append/3*), has a weight of 1, and so *append/3* has a weight of 1. The second goal for *nrev/2* is also recursive, so *nrev/2* is considered to have weight 1; but it also generates an *append/3* goal that has weight 1, and so *nrev/2* is finally given a weight of 2.

The main criticism of this approach is that it is too simple. It does not take into account the relationships between variables in clauses; for example, dependencies between goals and the size of data passed between goals on variables (streams).

3.4.3 Weights or No-weights

Using the ideas from Tick and Crammond, we have two different ways of ordering goals in the local goal queue:

Weights goals are ordered by Tick weights;

No-Weights goals are executed in a left-to-right depth-first manner.

Part of our investigation is to see whether one of these goal ordering strategies gives better program execution performance than the other. Does ordering the goal queue

```
local_schedule(Goal){
  goal_queue_add_front(Goal);
}

local_deschedule(){
  return goal_queue_remove_front();
}

remote_schedule(To){
  message(goal_tell(To,myid,Goal));
}

remote_deschedule(){
  To = choose_agent();
  message(goal_ask(To,myid));
}
```

Figure 3.2: Goal queue interface definitions to implement the local depth-first execution, breadth-first distribution strategy.

by weight significantly reduce execution time or does the overhead in implementing it outweigh any benefits? Is the weighted scheme too simplistic, or does it work better for some programs than for others?

3.4.4 Definitions

Here we give the goal queue interface definitions for the weights/no-weights strategies.

Figure 3.2 shows definitions to implement the no-weights strategy; that is, a local depth-first execution, breadth-first distribution strategy.

The local end of the goal queue (designated the front) is used like a stack where new goals are pushed on and goals are popped off for reducing. It should be remembered that with a stack goals will be popped off in the opposite order to which they are pushed on; to preserve the execution ordering of goals in the source program it must be arranged that goals are pushed onto the local end of the goal queue in the reverse order that they appear in the source program.

Scheduling a goal on a remote agent consists of sending it in a message to that agent, and descheduling a goal from another agent consists of sending a request for a spare goal from some agent. The procedure `choose_agent` defines how the scheduling agent will choose an agent to ask for spare work from all those in the system. This choice could

```
message_handler(Message){
  case(Message){
    .
    .
    .
    goal_tell(To, From, Goal):
      {
        goal_queue_add_front(Goal);
      }

    goal_ask(To, From):
      {
        if (goal_queue_length() > THRESHOLD)
          Goal = goal_queue_remove_back();
          message(goal_tell(From, myid, Goal));
        else
          message(goal_ask_nak(From, myid));
      }

    goal_ask_nak:
    .
    .
    .
  }
}
```

Figure 3.3: A definition of the inward message processing part of the communications agent.

be to choose the agent with the least load calculated from expected load information stored about some of the agents. Another possibility might be to choose agents that are physically close to minimise communication distances and hence message delay times. Or yet another example would be to allow an idle agent to choose at random any other agent in the system.

The part of the system that distributes goals to other agents is part of the communications manager in the message handler since distribution of goals is initiated on receipt of a `goal_ask` message.

A pseudo code definition of the part of the message handler we are interested in appears in figure 3.3. When a `goal_ask` message is received the length of the goal queue is checked against a threshold value. If the queue length is greater than the threshold then a goal is removed from the remote end of the goal queue (designated the back of the queue) and is sent in a `goal_tell` message to the requesting agent. If the threshold

```
local_schedule(Goal) {
  Weight = weight_of(Goal);
  goal_queue_insert_by_weight(Goal, Weight);
}
```

Figure 3.4: Modification to goal queue interface definitions to incorporate Tick's goal ordering strategy.

has not been exceeded, however, then a `goal_ask_nak` message tells the requesting agent to try another agent or try again later.

Figure 3.4 shows the alternative definition of `local_schedule` in the goal queue interface needed to implement goal ordering by weights. When a goal is put onto the goal queue then first of all its weight is found and then it is inserted in the goal queue according to that weight.

3.4.5 All-processors or nearest-neighbours?

One of the questions in designing a demand driven goal distribution strategy is, when a processing element has no work left, which processing agents should it ask for work?

We assume that the agents are connected together in some kind of network and that there is a notion of distance between agents. Agents that are next to each other for communications purposes are said to be neighbours.

We explore two ways of selecting an agent from which to ask for spare goals:

All-Processors: the idle processing agent asks any other agent;

Nearest-Neighbour: the idle processing agent asks one of the agents that are topological neighbours.

Figure 3.5 shows a pseudo code listing for a definition of `choose_agent()` for implementing the all-processors goal distribution strategy. The routine simply generates a random number, takes the modulus with respect to the number of agents, and returns the resultant identifier. If the identifier happens to be the same as the local agents' identifier then the calcualtion is repeated until an acceptable identifier is found and returned.

Similarly, figure 3.6 shows a pseudo code listing for a definition of `choose_agent` for implementing the nearest-neighbour goal distribution strategy. It is assumed that the identifiers of nearest-neighbouring agents are collected together in an array indexed

```
choose_agent(){
   do{
     ID = random() mod total_no_of_agents;
   }
   while(ID==myid);

   return ID;
}
```

Figure 3.5: Possible code for choosing an agent at random from all agents.

```
choose_agent(){
  ID = neighbours[myid][random mod no_of_neighbours(myid)];

  return ID;
}
```

Figure 3.6: Possible code for choosing an agent at random from neighbouring agents.

on agent identifiers and an integer. Choosing an agent is then done by generating a random number based on the number of neighbours the local agent has and then indexing on the array with the local agent's identifier and the generated numbers. The resulting identifier is then returned.

3.4.6 Summary of goal distribution strategies

We have described the two dimensions of goal distribution strategies we will explore. This provides the four possible strategies that have been explored:

AP-NW All-Processors, Non-Weighted: an idle processing agent may ask any other processor in the system for spare work and goals are executed locally in a depth-first manner.

AP-W All-Processors, with Weights: an idle processing agent may ask any other agent in the system for spare work and goals are ordered by weight.

NN-NW Nearest-Neighbours, Non-Weighted: idle agents ask only nearest neighbours for spare work and goals are executed locally in a depth first manner.

NN-W Nearest-Neighbours, with Weights: idle agents ask only nearest neighbours for spare work and goals are ordered by weight.

3.5 Apparatus

Our FGHC systems was desiged for a commercially available multiprocessor, a Meiko Computing Surface. A Computing Surface consists of a number of T800 Transputers connected to a reconfigurable switch network. This switch makes a variety of network topologies available to the user.

A T800 Transputer is a single chip processing element. It contains a 32 bit CPU, a floating point processor, a small amount of memory, and 4 serial communication links. The serial link of one Transputer may be connected to a similar link on another Transputer so that the two microprocessors may pass messages. With 4 links per Transputer, a natural topology is a 2-D mesh: each Transputer connects to four others using its serial links.

The Computing Surface does not connect the Transputers directly together but indirectly through a number of switching elements. These switches allow various different topologies to be configured, for example, pipelines, trees and toruses.

For our experiments we used a 2-D mesh topology. This is in part because it is a natural way to connect Transputers together, but there are also sound reasons why a mesh topology is attractive; we gave an argument for mesh topologies in the previous chapter. In summary, mesh topologies require a fixed number of communications links per processor and so are easily scalable to large numbers of processors. There is a penalty to pay in terms of the time taken to send a message between the two furthest nodes in the mesh, when compared to say a hypercube topology, but with the use of wormhole routing algorithms, this penalty should not be significant.

3.6 Experiments

This section starts by describing the experimental procedure followed when performing experiments to evaluate the goal distribution strategies detailed above. Then the test programs that were used are described.

3.6.1 Experimental procedure

The goal distribution strategies that we are interested in have been detailed in previous sections. To evaluate these strategies a number of test programs are executed on a Flat Guarded Horn Clause implementation (described in the next chapter) which executes on a T800 Transputer array configured as a 2-D mesh. Various statistics are gathered as a result of these executions and are analysed to show various aspects of system

```
for program in {hanoi, fib, qsort, primes, queen-ls}
  for sched in {AP-NW, AP-W, NN-NW, NN-W}
    for N in {1,2,4,9,16,25,36}
      for i in {1..10}
        execute system sched N program
```

Figure 3.7: Experimental procedure

performance. (The statistics and analyses are described later in this chapter.)

The experimental procedure is probably expressed best as preudo-code, given in figure 3.7. That is, a test program is chosen in turn from {*hanoi, fib, qsort, primes, queen-ls*}, a goal distribution strategy is chosen in turn from {*AP-NW, AP-W, NN-NW, NN-W*}, and each incarnation of the test program and goal distribution strategy is executed 10 times on from 1 to 36 processors. Each incarnation is executed 10 times so that the results may be averaged over the 10 executions and so give a more representative account of system performance.

Notice that the number of processors allowed is restricted to the numbers 1, 2, 4, 9, 16, 25, and 36 only since balanced square meshes can be made using these numbers. If intermediate numbers are used then arbitrary meshes can be constructed. To make the matter less complicated the experiments are limited to square mesh configurations only.

3.6.2 Benchmark programs

We have chosen the programs *hanoi, fib, primes, qsort*, and *queen-ls* as benchmark programs for evaluating the effectiveness of our goal distribution strategies. These are common benchmark programs that many researchers use for such purposes.

hanoi is an implementation of the Towers of Hanoi problem, *fib* is an implementation of a Fibonacci number generator, and *qsort* is an implementation of the quicksort algorithm.

These programs belong to the class of programs called *divide-and-conquer* programs; the problem to be solved is recursively split into a number of similar small versions of the original problem which can be executed in parallel.

Although these programs belong to the same class of programs it is worth pointing out the important features of their execution to help us understand the results of our experiments.

```
:- weight(hanoi/1,2).
:- weight(move/4, 2).

main :- hanoi(15).

hanoi(N) :- move(N, left, center, right).

move(0, _, _, _).
move(N, A, B, C) :- N =\= 0, M is N-1 :
  move(M,A,C,B),
  move(M,C,B,A).
```

Figure 3.8: Program listing of *hanoi*

```
:- weight(fib/2, 2).
:- weight(add/3, 0).

main :- fib(20, _).

fib(0, R) :- true : R = 1.
fib(1, R) :- true : R = 1.
fib(N, R) :- N>1, N1 is N-1, N2 is N-2 :
  fib(N1, R1), fib(N2, R2), add(R1, R2, R).

add(R1, R2, R) :- S is R1+R2 : R=S.
```

Figure 3.9: Program listing of *fib*

A listing of the *hanoi* program is given in figure 3.8. The bulk of the computation of *hanoi* consists of a recursive call to *move/4*, which when called creates two *move/4* subgoals. Executing *hanoi* will result in a balanced binary computation tree. Another feature of *hanoi* worth noting is that the branches of the tree are independent and so no binding messages or suspensions should happen during execution. This version of the hanoi program was directly translated from a Prolog equivalent used by Verden [Ver91] as a benchmark program.

A listing of the *fib* program is given in figure 3.9. The computation tree for *fib* is similar to that of *hanoi*, in that the bulk of the computation comes from the recursive *fib/2* relation. On execution of *fib* a ternary tree is formed where the major branches are the *fib/2* branches and one level *add/3* branch. The important differences between *fib* and *hanoi* are that the branches in *fib* are not balanced and that the *add/3* branches depend on results from the *fib/3* branches. This means that execution of *fib* will likely

```
:- weight(go1024, 3).
:- weight(qsort/3,3).
:- weight(part/4, 1).
:- weight(list/2, 0).

main:- go1024(_).

qsort([],Rest,Ans) :- true : Rest=Ans.
qsort([X|R],Y,T)   :-
  part(R,X,S,L), qsort(S,Y,[X|Y1]), qsort(L,Y1,T).

part([X|Xs],A,S,L) :- A<X  : L=[X|L1], part(Xs,A,S,L1).
part([X|Xs],A,S,L) :- A>=X : S=[X|S1], part(Xs,A,S1,L).
part([]    ,_,S,L) :- true : S=[], L=[].

go1024(_) :- list256(L,L2),list256(L2,L3),
             list256(L3,L4),list256(L4,[]),
             qsort(L,A,[]).

list256(L,E) :-
  L = [128, ..., 239,241,243,245,247,249,251,253,255|E].
```

Figure 3.10: Program listing of *qsort*

result in suspensions and transmission of bindings between processors.

The program listing for *qsort* is given in figure 3.10. The computation tree for *qsort* is again similar to that of *hanoi* in that a binary tree of *part/4* goals is evaluated. There are two major differences between *qsort* and *hanoi*. Firstly, the computation tree is unlikely to be a balanced binary tree since the shape is dependent on the data to be sorted. And secondly, there are data dependencies between the *part/4* goal and the recursive *qsort/3* goals in the second clause of *qsort/3*, whereas in *hanoi* all goals are independent. Because of these differences *qsort* is harder to squeeze speedup out of than *hanoi*. The main problem is that the two *qsort/3* body goals are fed by a *part/4* body goal. If we assume that all three body goals execute at roughly the same speed then the *part/4* is likely to dominate the computation and limit performance.

The last two programs, *primes* and *queens-ls*, belong to the *generate-and-test* class of programs: a goal generates potential solutions on a stream, and non-solutions are filtered out by filter goals.

primes is an implementation of a well known algorithm for finding prime numbers. This is done by generating a list of consecutive integers and then testing the integers for primeness with a chain of filter goals. On execution *primes* produces an unbalanced computation where goals communicate with each other substantially. A feature of primes is that as the number of primes increases, the filter processes further away from

```
:- perpetual gen/3.

:- weight(primes/2, 2), weight(sift/2,   2).
:- weight(filter/3, 1), weight(filter/5, 1).

main :- primes(800,_).

primes(S,X) :- true : gen(2,S,A), sift(A, X).

gen(Max, Max, Out) :- true : Out = [].
gen(N, Max, Out) :-  N < Max, N1 is N + 1 :
  Out = [N|Out1],
  gen(N1, Max,Out1).

sift([], R) :- true : R=[].
sift([H|T], R) :- true :
  R = [H|R2],
  filter(H, T, R1),
  sift(R1, R2).

filter(_, [], R) :- true : R = [].
filter(P,[H|T], R):- M is H mod P : filter(M, P, H, T, R).

filter(0, P, H, T, R) :- true : filter(P, T, R).
filter(M, P, H, T, R) :- M =\= 0 :
  R = [H|R1], filter(P, T, R1).
```

Figure 3.11: Program listing of *primes*

the integer generator filter out less numbers than those nearer to the integer generator. The higher the prime number then the less filtering the appropriate filter process will do and the more it will be suspended waiting for the next potential primes to be passed to it.

queen-ls is an all solutions implementation of the well known N-queens problem: *how to place N queens on a N-by-N chess board such that no queen attacks any other queen.*

This particular implementation is very similar in design to the *primes* program; partial solutions to the problem are generated and useless solutions are removed by filter goals. The major difference in style is that *queen-ls* uses the layered streams technique. So although *queen-ls* will share some execution characteristics with *primes* we might expect some differences due to the use of layered streams [OM87] in *queen-ls*. Instead of a pipeline of filter processes *queen-ls* generates a tree of filter processes. Each filter processes is incrementally testing a possible solution. Shared memory implementations have shown *queen-ls* to be highly parallel and good speedups can be obtained[Cra88]. In a distributed memory environment, where communicating values on streams between goals on different processors can be costly, such good speedups are not likely to be attained.

Listings of the programs are given again in appendix A.

3.7 Measures

In this section we describe the information that is collected during an execution of our system and the measurements that are calculated from that information.

During a single run of the system the following information is gathered:

Execution time: This is calculated as the time from when the initial goal is put on a run queue for execution until the last goal in the system has finished executing.

Reductions count: Each processor counts the number of reductions it has made during system execution.

Suspension count: Each processor counts the number of suspensions made during system execution.

Binding asks sent: A count is made of the number of binding ask messages sent on each processor during program execution.

Goal asks sent: A count is made on each processor of the number of goal asks it has sent during program execution.

```
:- perpetual count/2, count/3.

:- weight(go/3,       50), weight(queen/4,    35).
:- weight(q/4,        25), weight(filter/4,   15).
:- weight(fromLStoL/2,20), weight(fromLStoS/4,20).
:- weight(count/2      0), weight(count/3,     0).

main :- go(8,_,_).

go(M,N,A) :- true |
  queen(1,M,begin,A), fromLStoL(A,B), count(B,N).

queen(I,N,In,Out) :- I =< N, I1 is I+1 |
    q(1,N,In,In1),
    queen(I1,N,In1,Out).
queen(I,N,In,Out) :- I > N | Out = In.

q(I,N,In,Out) :- I =< N, I1 is I+1 |
    Out = [[I|In1]|Out1],
    filter1(In,I,1,In1),
    q(I1,N,In,Out1).
q(I,N,_,Out) :- I > N | Out = [].

filter1([[J|In]|Ins],I,D,Out) :-
    J =\= I, D =\= I-J, D =\= J-I, D1 is D+1 |
    Out = [[J|NewIn]|Out1],
    filter1(In,I,D1,NewIn),
    filter1(Ins,I,D,Out1).
filter1(begin,_,_,Out) :- true | Out=begin.
filter1([],   _,_,Out) :- true | Out=[].
/* otherwise. */
filter1([[_|_]|Ins],I,D,Out) :- true |
    filter1(Ins, I, D, Out).

fromLStoL(LayeredStream,List) :- true |
    fromLStoS(LayeredStream,[],List,[]).

fromLStoS([[A|LS1]|Rest],Stack,L0,L2) :- true |
    fromLStoS(LS1,[A|Stack],   L0,L1),
    fromLStoS(Rest,Stack,      L1,L2).
fromLStoS(begin,Stack,L0,L1) :- true | L0=[Stack|L1].
fromLStoS([],_,L0,L1) :- true | L0=L1.

count(L,N) :- true | count(L,0,N).

count([X|Xs],M,N) :- M1 is M+1 | count(Xs,M1,N).
count([],M,N) :- true | N = M.
```

Figure 3.12: Program listing of *queen-ls*

To summarise, at the end of a system run two types of information are output: the overall execution time for the system; and counts of the number of reductions, suspensions, binding asks messages sent, and goal ask messages sent for each processor.

From this information we obtain a number of useful measures:

- mean execution time
- mean speedup
- load balance
- mean total suspensions
- mean total binding requests
- mean total goal requests

3.7.1 Mean execution time

This gives a measure of the average absolute time taken for a set of system runs where the number of processors, program, and type of goal scheduler are kept fixed. If a system configuration is run m times and the execution time for a run of the system is t_i then the mean execution time is given by:

$$T_n = \frac{1}{m} \sum_{i=1}^{m} t_{in}$$

where t_{in} is the i-th run with n processors, and T_n is the mean execution time of the runs made with n processors.

For a good system we want to minimise the average execution time.

3.7.2 Mean speedup

Given the measure for mean execution time we can calculate a mean speedup figure which is:

$$E_n = \frac{t_{ref}}{T_n}$$

where n is the number of processors in the system configuration and E_n is the mean speedup of a series of runs of that configuration. The constant t_{ref} is the time taken

to execute the system on a single processor. With this measure we can compare the speedups obtained by system configurations with differing goal distribution mechanisms.

For a good system we want to maximise the mean speedup.

3.7.3 Load balance

A well balanced system should evenly distribute reductions across the processors of the machine. Given that the system gives a reduction count for each processor after a system run, a simple measure of load balance would be to calculate the standard deviation of the reduction counts. That is:

$$s_n{}^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (\bar{r}_n - r_{ni})^2$$

where s_n is the standard deviation for a system run with n processors, $\bar{r}_n$ is the mean number of reductions, and r_{ni} is the reduction count on the i-th processor[1].

Instead of using the standard deviation we use the coefficient of variance as a measure of load balance. The coefficient of variance is calculated as:

$$l_n = s_n / \bar{r}_n$$

The advantage of using the coefficient of variance is that data with differing scales can be compared. The mean number of reductions changes as we increase the number of processors for a fixed size program (with a fixed number of reductions), so using the coefficient of variance may allow us to compare the load balance values of runs of the system with different numbers of processors.

As with all the other measures, we take the average of the coefficient of variance over a number of runs of a particular system configuration. The overall value of load balance averaged over many runs is represented by L_n for n processors.

The lower the load balancing figure the better balanced the system. The aim of a good system might be to minimise L_n as long as execution time is reduced as a result.

3.7.4 Mean total suspensions

One of the measures of a system is the number of suspensions that are incurred during execution of a program. The total suspensions is calculated by summing the suspensions incurred on each processor after system execution.

[1] This can also be written as $s_n{}^2 = \frac{n\sum r_{ni}{}^2 - \left(\sum r_{ni}\right)^2}{n(n-1)}$

If several runs of a system configuration are made then a more typical value to use is the average of the total suspension counts of each run. This is called the mean total suspension measure and is represented by the term S_n when n processors are used for the system runs.

A good scheduler will minimise S_n since suspending and reawakening goals is costly.

3.7.5 Mean binding requests

Another measure of a system is the number of requests for variable bindings made during system execution. This is calculated in a similar way to the mean total suspension measure in that it is the average of the sum of total number of binding ask messages taken from several system runs. The mean binding requests measure is represented by the term BR_n.

A good scheduler will minimise BR_n since asking for and sending terms between goals is costly.

A simple way to minimise BR_n would be to execute the program on a single processor since then BR_n would be zero. Obviously it is good to minimise BR_n to the point where execution time is reduced rather than increased.

3.7.6 Mean goal requests

Another measure calculated in a similar way to mean total suspensions and mean binding requests is mean goal requests except that the total number of goal asks messages sent for each run is calculated and then the averages of these values taken. The mean goal requests measure is represented by the term GR_n.

A processor generally sends a goal ask message when it has no work left in its run queue, that is when it is idle, and so the number of goal ask messages sent during a system run reflects the amount of time processors have spent idling.

A good scheduler will minimise idleness and will minimise GR_n.

Again this could be achieved simplistically by executing the system on a single processor which may be counterproductive. The value of GR_n should be minimised as long as execution time is reduced rather than increased.

3.8 Summary

In this chapter we have detailed framework that we used to perform experiments with goal distribution strategies for the committed-choice languages.

We have chosen to design and implement an FGHC interpreter that executes on a Meiko Computing Surface, a T800 Transputer array with a reconfigurable topology. The computer is configured in a 2-D mesh topology and uses wormhole routing.

We have designed a processing agent model of a distributed language system with which we have described our goal distribution strategies.

The goal distribution strategies that we have chosen to investigate are:

AP-NW All-Processors, Non-Weighted: an idle processing agent may ask any other processor in the system for spare work and goals are executed locally in a depth-first manner.

AP-W All-Processors, with Weights: an idle processing agent may ask any other agent in the system for spare work and goals are ordered by weight.

NN-NW Nearest-Neighbours, Non-Weighted: idle agents ask only nearest neighbours for spare work and goals are executed locally in a depth first manner.

NN-W Nearest-Neighbours, with Weights: idle agents ask only nearest neighbours for spare work and goals are ordered by weight.

A significant part of the chapter was to describe measures by which we can compare various aspects of the performance of the goal distribution strategies:

- average execution time;
- average speedup;
- load balance;
- average suspensions;
- average number of binding requests;
- average number of goal requests.

Chapter 4

Interpreter: Description and Implementation

4.1 Introduction

In this chapter we describe our FGHC system that was used to experiment with goal distribution strategies.

The system was designed and implemented from scratch but many of the techniques used are inspired by descriptions of implementations by other researchers; in particular, our interpreter is influenced by the description of Prolog interpreter design in Maier&Warren[MW88], by the parallel Flat Concurrent Prolog implementation by Taylor *et al*[TSS87], and the shared memory implementation of Crammond[Cra88].

Firstly, we give an overview of the interpreter we have implemented and describe how it has been adapted for implementation on a distributed memory parallel computer.

Secondly, we give a description of the target computer system: its hardware components and communications software.

Thirdly, we give a detailed description of our interpreter implementation for our target hardware in terms of the data structures used and the agents that manipulate them.

Fourthly, we give a pseudo code description of the routines used to implement the goal distribution strategies we will experiment with.

Lastly, we give performance measurements for the single processor configuration of the system for a variety of benchmark programs and we analyse the parameters of the single node software system showing how they affect the system performance.

In the next chapter we present the results of the experiments conducted with the interpreter.

4.2 The Interpreter Overview

We have adopted a process pool based model for the interpreter. This interpreter has a number of elements:

Clause Store holds the clauses of the program against which goals are matched for execution.

Registers are used to hold temporary values during a reduction, for example, they are used to hold the arguments of a goal during its reduction.

Heap is used to construct new terms during a reduction.

Process Pool is a collection of processes where a process represents a goal queued for reduction.

Unification Stack is used when matching or unifying two terms against each other.

Suspension Stack is used to hold the variables that a process has suspended on during a reduction. If the entire reduction suspends then the process is suspended on the variables in the stack. That is, for each variable on the Suspension Stack a Suspensions List is attached to it, holding a list of processes suspended waiting for the variable to become bound. If it becomes bound before program execution terminates then the processes in the Suspension List will be placed back into the Process Pool.

Reduction Agent is the agent that performs the reduction of a process, using the clauses in the Clause Store, to a number of sibling processes that are then added to the Process Pool.

See figure 4.1 for a diagram of the data structures present in the interpreter model.

4.3 Single Processor Interpreter Model

Above we have described the basic overall interpreter model. When implemented on a distributed memory parallel computer the Heap and Process Pool are distributed across the machine. Each processor executes a copy of the interpreter described above and

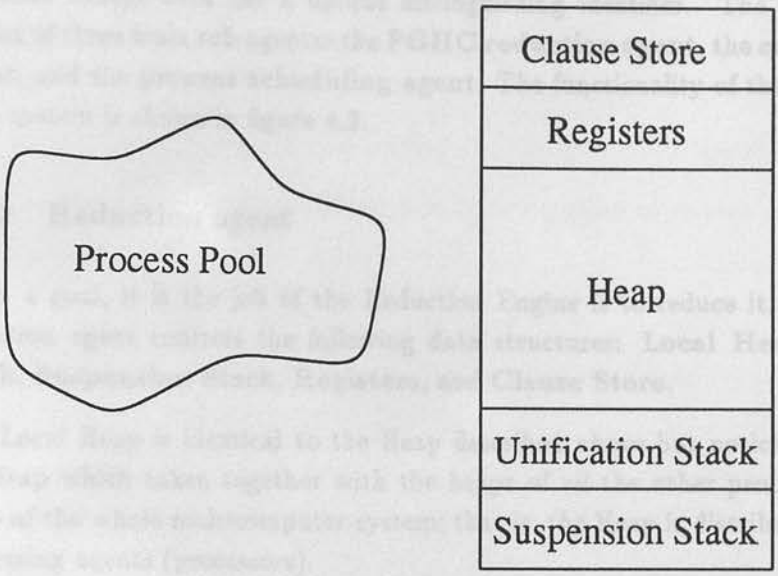

Figure 4.1: Elements of the FGHC interpreter

controls the portion of the Heap and Process Pool distributed to it. The interpreters execute in parallel and synchronise by passing messages to each other.

Here we relate the model of the interpreter described above using the processing agent model described previously in section 3.2.

4.3.1 Processing agent model

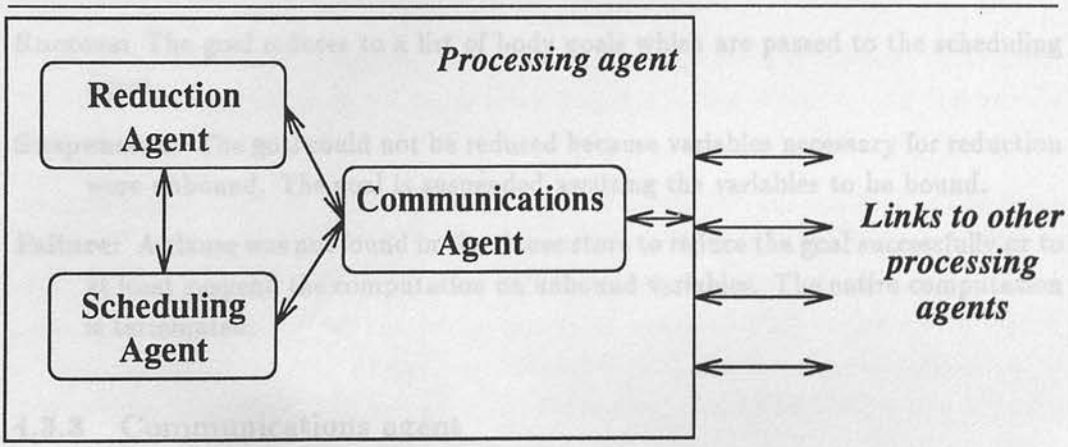

Figure 4.2: Functionality of the single processor system

The single processor interpreter is a **processing agent**; an identical agent on each processor except each has a unique distinguishing identifier. The processing agent consist of three main sub-agents: the **FGHC reduction agent**, the **communications agent**, and the **process scheduling agent**. The functionality of the single processor node system is shown in figure 4.2.

4.3.2 Reduction agent

Given a goal, it is the job of the Reduction Engine is to reduce it. To do this, the reduction agent controls the following data structures: **Local Heap**, **Unification Stack**, **Suspension Stack**, **Registers**, and **Clause Store**.

The Local Heap is identical to the Heap described above but notice that it is a Local Heap which taken together with the heaps of all the other processors forms the Heap of the whole multicomputer system; that is, the Heap is distributed between the processing agents (processors).

We distinguish these different forms of heap by using the terms: **local heap** for the part of the Heap held by a processor we are considering; **remote heap** for the part of the Heap controlled by a processor different to the one we are considering; or **global heap** for the whole distributed heap taken as one entity.

The Unification Stack, Suspension Stack, Registers, and Clause Store are identical to those for the overall system except that they are accessible only by the local processing agent.

Given a goal, the reduction agent attempts to reduce it to sub-goals. A single reduction step of a goal results in one of three outcomes:

Success: The goal reduces to a list of body goals which are passed to the scheduling agent.

Suspension: The goal could not be reduced because variables necessary for reduction were unbound. The goal is suspended awaiting the variables to be bound.

Failure: A clause was not found in the clause store to reduce the goal successfully or to at least suspend the computation on unbound variables. The entire computation is terminated.

4.3.3 Communications agent

It is the job of the communications agent to form the interface between the local processing agent and remote processing agents.

The communications agent controls three new data structures:

Message Buffers are used to hold messages for sending and receiving from remote agents.

Transmit Queue is used to hold message buffers queued for sending;

Receive Queue is used to hold message buffers queued to receive a message.

There are basically three types of message:

goal messages are to do with the management of processes: sending out goals and sending out requests for goals.

binding messages are to do with accessing variables controlled by remote processing agents: obtaining the value of a remote variable or binding a remote variable to a value.

system messages are to do with controlling the overall execution of the program: starting, stopping, and aborting the program.

4.3.3.1 Messages about bindings

Since the Heap is distributed over the memories of individual processors, it is necessary for a processing agent to be able to access and bind the value of remote heap cells.

Accessing the value of a cell happens during head matching or guard evaluation. If the cell resides on a remote heap then a message is sent to ask the remote processing agent to send back the value of the desired cell.

Binding the value of a cell happens during an output unification in the body of a clause. Again, if the cell resides on a remote heap then a message is sent to tell the remote processing agent to bind the cell to its value.

We have adopted a Saraswat-like[Sar89] notion of constraints on variables in our Heap model. That is, an uninstantiated variable stands for a Heap cell that has no constraints imposed on it, and a bound variable is a Heap cell that is constrained to a particular value. Examining a cell value is performed by an $ask(Var, Val)$ action where Var is a reference to the cell, and its value is returned in Val. Similarly, a cell referenced by Var can be constrained to a value Val by performing a $tell(Var, Val)$ action.

For the case where the processing agent is examining/constraining the value of a cell on a remote heap, we introduce messages for performing remote *ask* and *tell* operations. The messages are **binding_ask(T,F,Var)** and **binding_tell(T,F,Var,Val)**

respectively, where **T** is the identifier of the agent to which the message is sent, **F** is the identifier of the agent from which the message originated, **Var** is a variable reference, and **Val** is a term (which can be another variable reference).

For example, if agent A_1 requires the value of a heap variable Var held by agent A_2 it would send a *binding_ask*(A_2, A_1, Var) to agent A_2. This can have one of three results:

1. If Var on A_2 is instantiated with the value of Val, then Val would be sent back to A_1 by sending a *binding_tell*(A_1, A_2, Var, Val).
2. If Var were to dereference to an uninstantiated local variable, say Var_l then a **broadcast note**, $bn(A_1, Var)$, is attached to Var_l. The broadcast note means: *When this variable becomes bound send its value back to agent* A_1. That is, if Var_l is subsequently bound to some non-variable[1] Val then a *binding_tell*(A_1, A_2, Var, Val) will be sent to agent A_1.
3. If Var were to dereference to an uninstantiated remote variable, say Var_r on agent A_3, then a broadcast note, $bn(A_1, Var)$, is attached to the copy of Var_r on the local processor, and a *binding_ask*(A_3, A_2, Var_r) is sent to agent A_3.

   If Var_r on agent A_3 subsequently becomes instantiated to some non-variable[1] Val, then A_3 will send a *binding_tell*(A_2, A_3, Var_l, Val) message to A_2, which will cause Var_l to be bound to Val, the binding note will be picked up, and then in turn agent A_2 will send a *binding_tell*(A_1, A_2, Var, Val) message to agent A_1.

When a processor receives a *binding_tell*(P_n, Var, Val) then the dereferenced local value of Var, say Val_l, is output unified with Val, that is a $Val = Val_l$ is performed. This may cause more *binding_tell*s to be sent if remote variables in subterms of Val or Val_l become bound.

4.3.3.2 Messages about processes

Messages about processes are used to redistribute goals between the local process pool and remote process pools of other processing agents. There are four messages (**goal_ack** has been added to the description in the previous chapter):

goal_tell(T,F,Pid,P) is for sending the goal P, from agent F, to the remote agent T and it is from the agent F. Pid is a unique identifier for P assigned by agent F.

[1] A *binding_tell* message is sent if Var is bound to a non-variable: an integer, constant, or structure. If it is bound to a variable then the resulting actions depend on whether the variable is remote or local, and if remote whether it resides on a processor with identifier greater or smaller than the local one. To keep the examples simple we restrict them to the non-variable case.

goal_ask(T,F) is sent to the agent with identifier T and is asking for a spare process from the agent on that processor. It is from the agent F.

goal_ask_nak(T,F) is sent to the agent T telling it that the agent F has no spare processes.

goal_ack(T,F,Pid) is sent to agent T from agent F to indicate to agent T that agent F completed execution of goal Pid.

An idle agent may try to steal a goal from another agent by sending it a *goal_ask* message. For example, to ask for a goal from agent A_2, agent A_1 would send a message $goal\_ask(A_2, A_1)$ to agent A_2.

In response, agent A_2 performs one of two actions:

1. if A_2 has a spare goal then it may send it to A_1 by replying with a $goal\_tell(A_1, A_2, Pid, G)$ where G is the goal.
2. or if A_2 has no spare goals then it would reply with a $goal\_ask\_nak(A_1, A_2)$ causing agent A_1 to try elsewhere for more work.

(It is also possible for an agent to spontaneously send a goal to another processor by sending an unsolicited *goal_tell* message.)

Once an agent has received a process from a remote process pool and placed it in its local process pool, it can then reduce it to sibling processes using the Clause Store. If the original process and all its siblings eventually terminate then a $goal\_ack(A_1, A_2, Pid)$ message is returned to the agent that initiated the $goal\_tell(A_2, A_1, Pid, G)$. This is done so that the overall system can keep track of outstanding goals, tell when they have terminated, and so determine when the program has terminated.

4.3.3.3 Messages about the system

The last set of messages are system control messages: to start, stop, and abort the system. These messages are sent either to or from a distinguished agent, the **master**, which controls the overall execution of the system. In other respects the master is exactly the same as any other processing agent.

task_start(T,M) is sent to agent T to tell it that program execution has started. This message is sent from agent M, the master.

task_term(T,M) is sent to the agent T from the master to inform it that the program has terminated.

4.4.1 Hardware

The machine that we implemented for is a Meiko Computing Surface consisting of an array of Inmos T800 Transputers. We used the Edinburgh Concurrent Supercomputer(ECS) on which one can usefully execute programs on between 1 and 130 Transputers.

A T800 Transputer is a 'single-chip processor' in that it has all the necessary functionality to build a computer system. In particular is has a 32 bit CPU with a floating point co-processor, four asynchronous bidirectional serial communications links for connecting to other Transputers/devices, and a small amount of on-chip memory (4 kbytes). For realistic general purpose computing, the Transputers in the ECS have external memory; usually 4 Mbytes per processor although some have 16 Mbytes.

In the ECS, the communications links of the Transputers are fed into a reconfigurable switch which allows the processors to be connected into various topologies. When executing, each processor is allocated a unique integer which is the identifier for that processor. The first processor has identifier 0, the next, identifier 1, and so on. The identifier is available to the software executing on the processor by a call to a system routine.

The runtime configuration of the processor is specified in a configuration file. This specifies which processors will have their communications links connected to each other (via the reconfigurable switch) and which object code modules will execute on which processors.

In our FGHC system the processors are arranged in a 2-D square mesh and each processor executes an identical FGHC processing agent.

4.4.2 Software and communications environment

The interpreter is written entirely in the C programming language with Meiko's CSTools extensions for sending messages between processors.

CSTools provides a **global transport space** over the target machine. A **transport** is like a mailbox through which messages can be sent and received.

A processor declares a transport in the global transport space and attaches a symbolic name to it. Another processor, if it knows the symbolic name, can then pick up a reference to the transport by performing a transport lookup in the transport space. The processor can then send a message on the transport, and the processor that declared the transport can receive the message. Like a mailbox, many processors can pick up on the same transport and send messages to the processor that owns it.

In our system we use this CSTools mechanism in the following manner. Each processor declares a transport with a symbolic name attached derived from its processor number. Each processor can then look up the transports of each other processor and store them in an array indexed on processor number. In this way each processor now has a way of communicating with each other processor.

To send a message on a transport, CSTools provides a command **cs_send** which takes as parameters a reference to a transport, a character array in which the message is stored, and the message length. The message is then queued ready for transmission on the transport and the processor can resume execution.

Similarly, there is a **cs_recv** command for receiving messages. The asynchronous version of this command simply takes a character array and a transport as parameters. This array is then ready to receive the next message that arrives on that transport. Note that this means that the size of messages must be predetermined to be less than or equal to the length of the character array or else an exception occurs. This effectively means that messages have a maximum fixed size.

Because we are using the asynchronous sending and receiving communications mode, there is also a **cs_test** command to test if a certain character array has had its message successfully sent to or received from a remote processor.

When sending messages CSTools uses a form of wormhole routing. This means that the time to send a message should be mostly dependent on the message length (for suitably long messages) rather than on the number of intermediate processors between the sender and receiver.

4.5 Implementation

Here we describe our implementation of the interpreter. There are many details that we have not covered in the processing agent model which are fully described in this section, such as how the systems detects termination, and how terms are packaged to be sent from a local processor to a remote processor. We will again follow the same structure as the interpreter description by describing the data structures and then the processing agents.

4.5.1 Data structures

The data structures are as follows:

- Heap;

- Registers;
- Suspension Lists;
- Unification Stack;
- Suspension Stack;
- Symbol Table;
- Code Area;
- Remote Binding Cache;
- Goal Queue;
- Process Records.

4.5.1.1 The Heap

The Heap is where terms are constructed and manipulated during the execution of a program. It is an array of Heap cells. Heap cells are addressed using a *heap index.* Cell elements are allocated from the bottom of the Heap and the Heap grows upwards. The top-of-heap index is held in the special register H.

Each cell consists of two parts: a *tag field* and a *data field.* In our implementation a cell is constructed from one 32 bit word of data since the T800 target processors have a word width of 32 bits. The tag is 3 bits wide and the data field is 29 bits wide.

There are six possible cell types which are:

tag	*data*
VAR	pid, heap index
SUSP	sl_index
STR	functor_id, arity
CON	functor_id, 0
SYS	builtin_id, arity
INT	integer

The **VAR** cell is used to implement a distributed Heap over a distributed memory machine. The *data* field contains a pointer into the Heap of the local processor or a remote processor. The pointer is formed from two subfields in the *data* field: *pid* and *heap index.* Each processor has a unique identifier. The value of the *pid* field is the identifier of the processor on which the referenced Heap cell resides. The *heap index* is the index number of the referenced cell on that processor.

The width of the two *data* subfields has implications for the organisation of the machine. The total *data* field is 29 bits wide. How the 29 bits are allocated to the subfields limits the number of processors and the number of Heap cells that the system can support. We chose to allocate 7 bits to the *pid* subfield, thereby supporting computer systems with up to 128 processors, and to allocate the remaining 22 bits to the heap index, thereby allowing the Heap on each processor to consist of a maximum of 4 Mcells (or 16 Mbytes of memory). For computer systems with more processors and/or more memory the allocations would need to be revised or two words used per cell.

An uninstantiated variable points to itself. That is, if a variable held at heap index 43 on processor number 27 is uninstantiated it would contain these values:

$$43 \rightarrow < VAR,\ 27,\ 43 >$$

A variable is tested to see if it is local to the processor by testing if its *pid* subfield is equal to the unique identifier of the processor.

Note that the VAR cell is the only one that has a *pid* field; it is the only cell type that may reference Heap cells on other processors. All other types of cell reference other data elements held locally.

To avoid referencing loops among variables there is a **binding convention**. If two local variables are to be bound together, then the variable with the higher index is made to point to the one with the lower index. Referencing loops may also occur across the different processors' Heaps. To avoid this there is a **global binding convention** whereby the variable with the higher *pid* field is made to point to the variable with the lower *pid* field. ([TSS87] supplied the idea for this binding convention.)

SUSP marks an uninstantiated variable on which processes or messages are suspended waiting for the variable to be bound before they can resume. The *data* field indexes into an array of Suspension Lists. The Suspension List the holds information about the processes and/or messages suspended waiting on the variable. Suspension Lists are explained in more detail below.

STR tags a cell that represents a user defined structure. The *data* field of the cell consists of 2 subfields: *functor_id* which is a unique integer for the functor of the structure; and *arity* which is the arity of the structure. A STR cell is followed by arity number of cells which are the arguments of the structure. The *functor_id* is an index into the Symbol Table (described below).

CON tags constants: functors with arity zero.

SYS is used to distinguish between a builtin term and a user defined term. In other respects the SYS cell behaves like a STR cell. That is the data field consists of an

identifier subfield, the *system_id*, and an *arity* subfield, which holds the arity of the builtin term. Since the user program which the system executes is constructed from Heap cells, it is important to distinguish between system defined terms and user defined terms. When coming to execute a goal it makes it possible for the interpreter to tell whether, in the case of a user defined term, to reduce the goal using the clauses in the Clause Store, or in the case of a system term, to call an appropriate system routine.

INT is used to hold an integer value, in this implementation a 29 bit integer value. This cell type enables the use of integer values for use in simple arithmetic operations.

4.5.1.2 Registers

The Registers are a special array of cells which are used to hold the arguments of the current goal being executed. Given that the current goal has N arguments, then before the reduction of the goal, the arguments are copied into the first N registers. If an argument is simple, such as an integer (INT) or a constant (CON), then the argument is copied directly into the register. If the argument is complex, such as a structure (STR), then a pointer is stored in the register, made from a VAR cell, which points to the place on the heap where the argument is stored.

Registers may also be used to hold temporary values used during the reduction of a goal.

By convention, register cells are allocated at a lower machine address than the Heap cells so that, during guard execution, unbound register cells can be identified as non-global variables and can thus be bound. Cells other than registers may not be bound during guard evaluation.

4.5.1.3 Suspension Lists

A Suspension List is used to annotate an uninstantiated variable with processes or messages that have suspended waiting for the variable to be bound. Suspension Lists are taken from a fixed array of Suspension Lists. The **sl_index** of a SUSP Heap cell points into this array. The array also forms a free list to allow the easy allocation and reclamation of suspension lists. The end of the free list is pointed to by the special register SLFL.

A suspension list is really two lists: **a process list**, and a **message list**. Both these lists are double ended lists so that it is a simple operation to concatenate two suspension lists; that may happen when two uninstantiated variables, with suspension lists attached to them, are unified with each other.

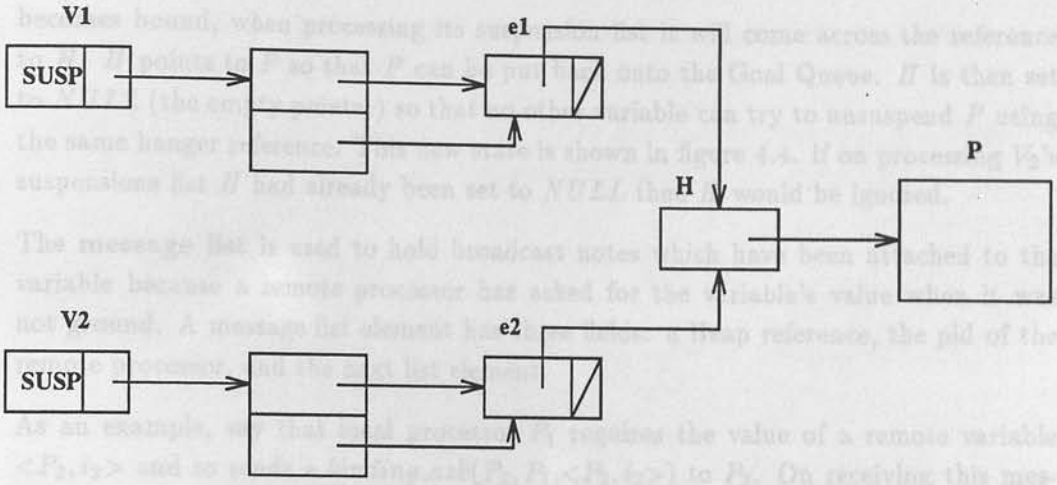

Figure 4.3: Suspension lists sharing a process record via a hanger.

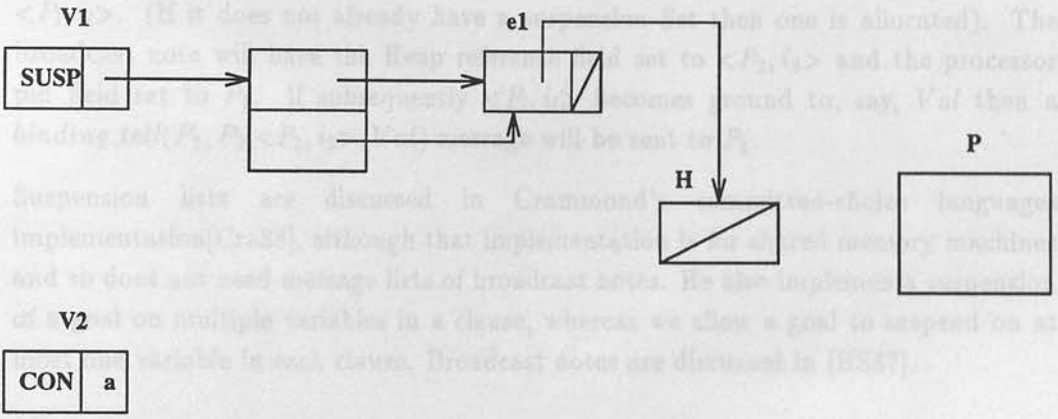

Figure 4.4: One suspension list is used to wake up the process record and the hanger is set to $NULL$.

The **process list** is a list of processes suspended on the variable that is referencing this Suspension List. The elements of the process list consist of two fields: an indirect pointer to a process via a **hanger**, and a pointer to the next element.

The hanger is used because a process may be suspended waiting on several variables to become bound but only one of the variables needs be bound to continue execution of the process. Hangers are described in [HS87] and [Cra88].

As an example, if during reduction, process P suspends on V_1 and V_2, a hanger H is created pointing to P. Process sublist elements e_1 and e_2 are created for the suspension lists of V_1 and V_2. e_1 and e_2 point to H. In this way V_1 and V_2 point indirectly to P through their suspension lists and through hanger H. This setup is shown in figure 4.3. Only one of the variables needs to be bound to continue execution of P. If, say, V_2

becomes bound, when processing its suspension list it will come across the reference to H. H points to P so that P can be put back onto the Goal Queue. H is then set to $NULL$ (the empty pointer) so that no other variable can try to unsuspend P using the same hanger reference. This new state is shown in figure 4.4. If on processing V_2's suspensions list H had already been set to $NULL$ then H would be ignored.

The **message list** is used to hold broadcast notes which have been attached to the variable because a remote processor has asked for the variable's value when it was not ground. A message list element has three fields: a Heap reference, the pid of the remote processor, and the next list element.

As an example, say that local processor P_1 requires the value of a remote variable $<P_2, i_2>$ and so sends a *binding_ask*$(P_2, P_1, <P_2, i_2>)$ to P_2. On receiving this message, P_2 may dereference $<P_2, i_2>$ to $<P_l, i_l>$ which happens to be an unbound variable local to P_2. In this case, a *broadcast note* is added to the suspension list of $<P_l, i_l>$. (If it does not already have a suspension list then one is allocated). The broadcast note will have the Heap reference field set to $<P_2, i_2>$ and the processor pid field set to P_1. If subsequently $<P_l, i_l>$ becomes ground to, say, Val then a *binding_tell*$(P_1, P_2, <P_2, i_2>, Val)$ message will be sent to P_1.

Suspension lists are discussed in Crammond's committed-choice languages implementation[Cra88], although that implementation is for shared memory machines and so does not need message lists of broadcast notes. He also implements suspension of a goal on multiple variables in a clause, whereas we allow a goal to suspend on at most one variable in each clause. Broadcast notes are discussed in [HS87].

4.5.1.4 Unification Stack

This is a special stack of cells used to hold cell values that are being recursively matched or unified together. The Unification Stack is used during guard execution for matching two terms and is used during body execution for output unifying two terms.

The unification stack has two registers associated with it: UST which points to the current top of the Unification Stack and USN which holds the number of elements on the Unification Stack.

4.5.1.5 Suspension Stack

This is another special stack of cells which is used to hold the values of variables on which the current process has suspended. When a processor suspends on a variable it is pushed onto the Suspension Stack. At the end of the attempted reduction of a

process, if the process has suspended overall, then the values on the suspension stack are popped off and used during suspension of the process.

Like the Unification Stack the Suspension Stack has two registers associated with it: SST, the top of the stack, and SSN, the number of elements in the stack.

4.5.1.6 The Symbol Table

Each functor in the program has a corresponding entry in the symbol table. Each symbol table entry has the following fields:

print name used to display the functor.

weight is used to hold the weight value of the functor.

first clause which is a pointer to the first clause in a linked list of clauses attached to this functor.

last clause is a pointer to the last clause attached to this functor.

The Symbol Table is used mainly for looking up the clauses for reducing a user defined process. The *functor_id* of a STR cell is an index into the Symbol Table which makes Symbol Table lookups simple. If the functor has no clauses attached to it then the first clause and next clause fields will contain the *NULL* pointer.

The contents of the Symbol Table are identical on each processor so that a functor transmitted from one processor to another will point to the same Symbol Table data on all processors. This is necessary for goal distribution to be possible.

4.5.1.7 Code Area and Representation

Since our system is an interpreter, code is stored along with data in the form of cells similar to Heap cells in the Code Area. There is an extra cell type used in the Code Area, the **REG** type. The data field of a REG cell holds an integer identifying the register to which it refers.

A clause is constructed from terms — a list of guard terms and a list of body terms. Guard terms will all be SYS's since only system calls may appear in the guard. Body terms will be either SYS's for system body calls or STRs for user defined body calls.

Clauses in the same relation are grouped together by forming them into a linked list of clauses referenced through the Symbol Table.

The *functor_id* field of a STR cell is also an index into the Symbol Table. The clauses referenced by a STR cell may therefore be picked up by using the *functor_id* field to look up the clauses in the Symbol Table.

As in the case of the Symbol Table, and for the same reasons, the Code Area on each processor must be identical.

4.5.1.8 Remote Binding Cache

A processor's Heap will eventually have variable cells which point into the Heaps on other processors. To get the value of a cell on a remote Heap it is necessary to send a message asking for it to the remote processor on which it resides.

The simplest approach would be to replace the remote variable reference with the received value. There may be, however, an identical remote variable on this processor, pointing to the same place on the remote Heap. It is likely at some point in the computation that this second variable will need to be related to its value. Under the simple scheme this would entail sending another message to the remote processor, obtaining another possibly identical copy of the value, and again replacing the instance of the remote variable with a copy of its value. If there are many such identical values then there will be many copies of the same data which will waste memory.

An alternative scheme, that we have adopted for this system, is to create a Remote Binding Cache to hold the values of remote variable bindings that a processor has asked for. The cache is a table of entries of the form <key, value> where key consists of the fields {pid, heap ptr} and value is an index into the local processor Heap.

If the value of a remote variable is needed, then firstly the Remote Binding Cache is checked to see if an entry already exists. If so then the value index is dereferenced and is used as the value of the remote variable. If no entry exists then a new entry is made in the cache with the *key field set* to the remote variable reference and the *value* field set to $NULL$ and a *binding_ask* message is sent to the remote processor.

When the value of a remote variable is received, the entry in the cache with *key* field matching the remote variable reference is found and the corresponding *value* field is dereferenced. The resulting Heap cell is then unified with the new value. In this way, multiple instances of remote variables will share the same physical data representing the value of the variable. This saves time in removing duplicate *binding_ask* and *binding_tell* messages and saves the memory that would be needed to duplicate copies of the value of a remote variable.

The disadvantage of this method is that it also complicates dereferencing since a remote variable needs to be checked against the Remote Binding Cache each time it is

dereferenced. So far we have not investigated the practical merits or otherwise of this approach.

The Remote Binding Cache is formed from a fixed array of key/value pairs. A simple hashing function is used to provide for near constant time access to the Remote Binding Cache. We have found that hashing on the *heap ptr* subfield of the variable proves adequate.

4.5.1.9 Goal Queue

The Goal Queue is a data structure holding the processes which are currently reducible. That is, it represents the process pool for the processor on which it resides and is therefore managed by the Scheduler module.

The Goal Queue is referenced through two registers: GQF which points to the first process in the Goal Queue, and GQB which points to the last process in the Goal Queue.

The Goal Queue is formed from a doubly linked list so that the scheduler has quick access to the front and back of the queue and can search the Goal Queue if need be from either the front or the back. This allows flexibility for the Scheduler to arrange the queue in a variety of ways.

4.5.1.10 Process Records

Each goal has a Process Record associated with it through which it is referenced. Process records have the following fields:

goal is a pointer to the goal stored on the Heap.

weight is the weight of the goal.

ancestor is a pointer to the process record of the goal that created this goal. Since goals may migrate across processors, the ancestor field has 2 subfields: *anc_pid* which contains the *pid* of the ancestor process record; and *anc_ptr* which contains a pointer to the ancestor process record on the processor referenced in *anc_pid.*

nchild contains the number of nonterminated outstanding sibling goals. When it has a value of zero all sibling goals have terminated allowing this goal to terminate also.

mode holds the state of the goal. A goal may be RUNNABLE, SUSPENDED, WAITING (for sibling processes to terminate), or TERMINATED.

next is a pointer to the next process record in a list of process records.

prev is a pointer to the previous process record in a list of process records.

With this process record structure a distributed AND-tree is built, reflecting the state of computation.

Process Records are allocated from a free list of process records. The special register PRFL is used to hold the front of the free list.

4.5.2 The Reduction Engine

Given a goal, it is the job of the Reduction Engine is to reduce it. A reduction consists of using the program represented in the Code Area to reduce the goal to a list of sibling processes. The Reduction Engine may not be interrupted/context-switched during a reduction; reductions are atomic.

A special register, CP, holds a pointer to the current goal. The goal is tested to see if it is a user defined predicate or a system builtin by checking the tag of the first cell of the goal field pointed to by the process record. If the tag is of type SYS then the *builtin_id* field is extracted, and the entire goal is passed to the appropriate builtin call. The system call will extract the arguments from the goal and act on them. If the system goal succeeds in reducing then its parent ancestor's *nchild* counter is decremented and the current process record is reclaimed.

If the tag is of type STR then the Reduction Engine is dealing with a user defined goal. In this case the arguments of the goal structure are loaded into the Register area. The Current Clause is selected by indirecting through the symbol table, using the *functor_id* field of the first cell of the goal field pointed to by the process record. A special register, CC, holds a pointer to the Current Clause being used to attempt the reduction.

4.5.2.1 Guard execution

The guard goal list is then extracted from the Current Clause. Guard goals are always builtins and so are executed as system goals in turn. There are three possible outcomes after attempting to execute the guard of the current clause:

Success The Reduction Engine carries on to the body process spawning phase.

Suspension One of the guard goals suspended on a variable; the variable is pushed onto the Suspension Stack. CC is set to the next_clause_of(CC) and guard exe-

cution is attempted again.

Failure One of the guard goals fails. The failure is noted and CC is set to the next_clause_of(CC) and guard execution is attempted again.

If all the guard goals of the clause have been attempted but none succeed then if at least one has suspended, the current process is suspended on the variables left on the Suspension Stack. The **mode** field of the current process is set to SUSPENDED.

If all the clause guards failed then the entire computation fails.

4.5.2.2 Body spawning

For each goal in the list of body goals for the clause pointed to by CC, a copy of the goal is made on the Heap, setting the arguments of the goal appropriately from the Registers.

A process record is created for the goal copy with the appropriate fields set: *ancestor* is set to point to the value of CP; *nchild* is set to zero; *mode* is set to RUNNABLE. In addition the *nchild* field of the current process record is incremented and its *mode* field is changed to WAITING since it is now waiting for the newly created child processes to terminate execution. Each new process record is handed to the Scheduler for adding into the Goal Queue.

There is a special case to the body spawning phase. If the current clause has no body goals — a unit clause — the *mode* field of the current process record is set to TERMINATED. The process record can now be reclaimed.

Note that clauses are attempted in sequence and that guard goals are executed in sequence. A single processing node executes a completely sequential model of an FGHC interpreter.

4.5.2.3 Last Call Optimisation

If the current process record is set to WAITING it is only being used as a placemarker in the AND-process tree; that is, only the *ancestor* field of the process record retains any use. Instead of allocating a new process record for each new body process, the last new body process can reuse the process record pointed to by CP; all the record fields are left untouched except that the *goal* field points to the new goal created on the Heap. This saves allocating one process record per reduction which saves both space and time.

4.5.3 Communications manager

The purpose of the communications manager is to handle messages between the other software modules — the reduction engine and the scheduler — and the network hardware.

4.5.3.1 Message Queues

The communications manager controls two queues: the transmission queue, pointed to by TQ, and the reception queue, pointed to by RQ. These queues are formed from lists of message buffers. Each message buffer has 2 parts: a header, and a body. The header consists of 5 fields:

type the type of the message used to decode the message body;

source the pid of the processor from which the message originated;

destination the pid of the processor for which the message is intended;

length1 the length of first field of the body;

length2 the length of second field of the body.

The message body consists of an array of characters. The information to be conveyed in the message is mapped onto this character array. Other than the header, the messages in our system need a maximum of two pieces of information per message carried in the body. *length1* and *length2* give the length of the two parts of the message in the body so that they may be extracted on receipt of the message.

Although messages may be of variable length, the size of the body of a message buffer is fixed. This is because a buffer used for receiving a message must have the space available to receive the message. The space must be allocated *before* the message is received and the space must be at least as big as the biggest message that could be received. This means a fixed message size for receiving which then puts a constraint on the size of messages in the system and fixes the size of buffers used for sending messages.

The size of buffer chosen has an effect on computation time depending on the how networking is implemented on the target machine. On a machine with wormhole routing, as in CSTools on MEIKO machines, the time taken to transmit the message is:

$$transmit\_time = startup\_time + f(length\_of\_message)$$

For efficient use of communications bandwidth messages should be long enough so that the *startup_time* does not dominate, but not so long that unnecessary load is put on the network.

Message buffers in TQ form a circular queue, awaiting transmission. Message buffers in the RQ also form a circular queue and wait for messages to be received into them.

Periodically RQ must be checked for incoming messages which must be processed. The processed message buffers are then requeued for receiving more messages. Similarly, TQ must be checked periodically for message buffers which have completed sending their message. Completed buffers can then be reused for sending subsequent messages.

The total number of buffers in TQ and RQ is selected at runtime and does not vary during a complete run of the system.

4.5.3.2 Message Types

There are three message types concerned with processes, bindings and system control.

There are four messages to do with processes:

type	field1	field2
goal_tell	process record	packed goal
goal_ask	—	—
goal_ask_nak	—	—
goal_ack	process record	—

There are two messages to do with bindings:

type	field1	field2
binding_tell	variable	packed value
binding_ask	variable	—

There are three control messages:

type	field1	field2
task_start	—	—
task_term	—	—
task_abort	—	—

4.5.3.3 Messages sent

Here we will detail the meaning of the above messages types and why they are sent.

goal_tell is used to send a goal from one processor to another. The body of the message contains a copy of the process record of the goal together with a representation of the goal packed from the Heap.

goal_ask is used to ask another processor for a spare goal. This message is sent when a processor has no goals in its local Goal Queue.

goal_ask_nak is used when a processor has received a *goal_ask* but has no goals to give away.

goal_ack is sent when a goal obtained from another processor has completed execution. The body of the message contains the process record of the goal. The ancestor information is used by the destination processor to update the process record of the ancestor.

binding_tell is used to send the binding of a variable to a remote processor. The body of the message contains the name of the variable, and the packed value of the variable. These are generated in response to a *binding_ask*.

binding_ask is used to ask for the value of a remote variable. The body of the message is the name of the variable wanted. These are generated, for example, when a goal suspends on a number of remote variables. A *binding_ask* is generated for each variable in the hope that an answer will free the suspended process.

task_start is sent by the master processor to tell a processor that program execution has started;

task_term is sent by the master processor to tell a remote processor that the computation is completed.

task_abort is sent to the master processor if an exception occurs from which the processor cannot recover, for example, lack of memory.

4.5.3.4 Messages received

Here we describe how a processor handles each type of incoming message.

goal_tell causes the process record to be scheduled onto the Goal Queue with the goal unpacked and added to the Heap of the receiver. The goal can now be executed by the processor.

goal_ask causes the communications manager to ask the Scheduler for a goal to be sent in reply. If there are no goals then a *goal_ask_nak* is returned. Otherwise an appropriate *goal_tell* is generated.

goal_ask_nak tells the processor that the processor it asked for a goal has none to give away. Another processor must be tried.

goal_ack tells the processor that a goal that it farmed out to another processor has terminated. Using the received process record the ancestor of the process is updated; its *nchild* counter is decremented and state changed.

binding_tell causes the processor to unify the variable held on the Heap with the value received. This may cause more *binding_tells* when unifying two terms containing remote variables.

binding_ask causes the processor to check if the variable asked for is bound or is unbound. If bound then a *binding_tell* reply is send giving the value of the variable. If unbound then a message annotation is made on the suspension list of the variable. If the variable is bound at some point then a *binding_tell* will be sent to the original requesting processor.

task_start causes the processor to start trying to reduce goals.

task_term causes the computation to terminate on the processor.

task_abort received by the master processor causes it to send task_kill messages to the other processors before terminating itself.

4.5.3.5 Packing/Unpacking of terms

Terms must be packed into a linear array of cells for transmission. We do this using a simple depth-first left-to-right packing algorithm until the maximum array size is met.

There is one problem with packing — how to pack pointers inside terms. That is, how to pack a term with arguments that point to other structures. This is solved by introducing a new cell tag type **REL**. This is used to make a pointer relative to its position in the message buffer. This adjustment is done on-the-fly as the term is being packed.

Unpacking a term simply consists of copying each cell of the message onto the Heap while translating REL cells back into VAR cells with the *heap index* field of the cell adjusted to the local Heap addresses.

4.5.4 The Scheduler

The Scheduler orders the Goal Queue and decides which goal should be executed locally next and which goal should be distributed globally next. To the other modules, the Scheduler offers calls to add or remove processes to the Goal Queue. These calls are:

local_schedule(P) adds goal P to the Goal Queue for execution locally;

local_deschedule()→P removes goal P from the Goal Queue for execution locally;

remote_schedule(Pid) schedules a goal remotely on the processor with identifier Pid;

remote_deschedule() makes steps to remove a goal from the Goal Queue of another processor.

We implemented the following distribution strategies:

AP-NW All-Processors (an idle processor asks any other processor for a spare goal), No-Weights (goals are ordered as they appear in the source program);

AP-W All-Processors, Weights (the goal queue is sorted on goal weights);

NN-NW Nearest-Neighbours (an idle processor asks a neighbouring processor for a spare goal), No-Weights;

NN-W Nearest-Neighbours, Weights.

The Goal Queue is accessed via the following routines:

gq_add_front(G) which adds the goal G to the front of the Goal Queue;

gq_remove_front()→G which removes the goal G from the front of the Goal Queue;

gq_add_back(G) which adds the goal G to the back of the Goal Queue;

gq_remove_back()→G which removes the goal G from the back of the Goal Queue;

gq_insert_by_weight(G,W) which adds the goal G into the Goal Queue using **insert sort** on the weight value W.

The first four routines are standard doubly linked list access routines. The last routine is used to order the Goal Queue by goal weight.

Figure 4.5 gives the definitions of the scheduler interface routines. There is a global variable `STRATEGY`, the value of which is the goal distribution strategy being used. If the strategy is a non-weights strategy then goals are scheduled locally by adding them to the front of the Goal Queue. As long as the goals are added in the reverse order to which they appear in the source program, when they are subsequently removed from the front of the queue for local execution, the goals will be executed locally in the order

```
local_schedule(P){
  case STRATEGY of {
  AP-NW || NN-NW:
    gq_add_front(P);

  AP-W || NN-W:
    W = weight_of(P);
    gq_insert_by_weight(P,W);
  }
}

local_deschedule(){
  return gq_remove_front();
}

remote_schedule(Pid){
  P = gq_remove_back();
  message(Pid, mypid, goal_tell(P);
}

remote_deschedue(){
  Pid = choose_processor();
  message(Pid, mypid, goal_ask());
}

choose_processor(){
  case STRATEGY of {
  AP-NW || AP-W:
    do {
      P = random() mod NP;
    } while (P!=mypid);

  NN-NW || NN-W:
    do{
     P = neighbours[mypid][random mod 4];
    } while (P!=dummy);
  }

  return P;
}
```

Figure 4.5: Definition of scheduler interface routines.

in which they appear in the source program (the front of the queue acts like a LIFO stack). The oldest scheduled goal will be at the back of the queue and that will be sent out for remote evaluation first.

If, on the other hand, the strategy uses goal weights then the goals are added to the Goal Queue according to weight using an **insert sort** algortithm. In this case, the goals with smallest weight will be at the front of the Goal Queue and will be executed locally first; goals at the back of the Goal Queue will have the highest weight and will be executed remotely first.

When a processor becomes idle it will perform a `remote_deschedule()` to obtain a goal from a remote processor. If `STRATEGY` is set to an all-processors strategy, then the processor is chosen randomly from all the processor in the system; `NP` is the number of processors in the system.

For nearest-neighbours strategies the chosen processor will be from one of the neighbouring processors. For this to be possible each processor must have the processor identifiers of its neighbouring processors. Distributed to each processor is a 2-D array, `neighbours`, which is indexed on processor identifier and integer. We can fix the dimensions of the array to be `NP`x4 since a processor can have a maximum of 4 neighbours. To choose a neighbouring processor, `neighbours` is indexed on the identifier of the local processor (`mypid`) and on a random number from 0...3. This processor identifier is used to send a message to obtain a spare goal. There is a special identifier `dummy` since not all processors have 4 neighbouring processors (processors at the edges have 2 or 3 neighbours); any empty spaces in the array are filled in with `dummy`.

4.5.5 The Toplevel

The Toplevel of the system coordinates the different modules. Its main task is to timeslice between the Communications Manager, to process incoming messages, and the Reduction Engine, to reduce processes. The ratio between communication and reductions can be critical since calling the Communications Manager when there are no incoming messages wastes time but reducing goals for too long can cause starvation of processes/bindings for other processors.

The Toplevel also keeps track of the local system state for its processor. The local processor state may be one of five values:

Running The Goal Queue has goals for execution;

Idle The Goal Queue is empty and no goals have been asked for from other processors;

Waiting The Goal Queue is empty and the system has asked for a goal from another

processor and is awaiting a reply;

Terminate Computation has terminated;

Abort An exception has occurred and computation has aborted.

```
toplevel(reductions_per_cycle){
  while{STATE != TERMINATE && STATE != ABORT}{
    if (!Is_Empty(GOAL_QUEUE)) STATE = RUNNING;

    case STATE of {
    RUNNING:
      Reduce(reductions_per_cycle);
        if (Is_Empty(GOAL_QUEUE)) STATE = IDLE;
    IDLE:
      Remote_Deschedule();
      STATE = WAITING;
    WAITING:
      donothing;
    TERMINATE:
      donothing;
    ABORT:
      donothing;
    }

    if (Process_Messages(RQ) == ABORT) STATE = ABORT;
    else Check_Messages(TQ);

    if (Has_Terminated()) STATE = TERMINATE;
  }
}
```

Figure 4.6: Pseudo code for toplevel module manager

The toplevel also periodically checks the message buffer queues TQ and RQ for any incoming communications or for any completed outgoing communications. Figure 4.6 gives pseudo code for the toplevel manager. Notice the parameter passed to Toplevel, `reductions_per_cycle`, which is subsequently passed to the call to the Reduction Engine, `Reduce()`. This parameter controls how many reductions are performed before control passes to processing the message queues RQ and TQ and so controls, in a very rough manner, the ratio between the amount of time spent reducing goals and the amount of time checking message buffers for new messages.

4.5.6 Start up and termination

So far we have neglected to mention how a program commences execution and how termination is detected. Although each processor executes an identical FGHC system loading identical Symbol Tables and Code Areas, one processor is marked as the Master Processor. The Goal Queue of the Master Processor is initialised with the Initial Process which represents the query goal. The Master Processor begins execution by sending *task_start* messages to each other processor, and then attempts to make the first reduction by reducing the Initial Process.

Termination is detected by the Master Processor when the *nchild* field of this Initial Process has the value 0. This will only happen when the other processes in the system have terminated. When this happens the Master Processor sends out *task_term* messages to each processor to indicate that the program has terminated. The processors then go into a phase of sending collected statistics to the master processor (described in the next section) before they terminate their local execution.

Note, however, that this will only detect termination. If at some point all processes in the system are suspended causing the system to *deadlock*, this deadlocked state will not be detected which is a common problem with distributed memory systems. Deadlock detection does not concern us in this thesis. We assume that the programs we will execute are guaranteed to terminate.

4.5.7 Collecting statistics

To collect execution statistics each processor has a number of counters:

reduction counter is incremented each time a goal is reduced to body goals;

suspensions counter is incremented each time a goal is suspended locally;

binding asks counter is incremented each time a binding_ask message is sent;

goal_ask counter is incremented each time a goal_ask message is sent.

After the program has terminated, each processor sends to the master its collected counter values. The master collects these together in a 2-D array indexed by processor number and type of counter. In addition, the master processor records the **execution time** which is calculated as the elapsed time from when the first *task_start* message is sent out and the first *task_term* message is sent out.

The master then dumps the array of counter values and the elapsed time at the end of program execution. This data dump can then be analysed later and performance

values generated using the measures we have proposed in the previous chapter (see section 3.7).

The master also calculates a raw LIPS rating: the sum of the reductions made by the processors divided by the execution time.

4.5.8 Main limitations of the system

It is worth mentioning those functions that would be part of a full system implementation but which we have left out of our development system. The most important functions that we have ignored are:

- garbage collection;
- I/O builtin calls.

A commercial system would need a garbage collector to recycle unreferenced Heap cells. We have chosen to ignore garbage collection because of the considerable implementation effort involved and because any garbage collection system, especially one with a dynamic component, would have unpredictable effects on our measurements of goal distribution strategies.

The second area of functionality that we have ignored is the provision of input and output primitives. Again we considered the provision of I/O as time consuming and irrelevant to the object of our study, that is, goal distribution strategies.

4.6 Single Processor Performance

Here we will give measurements for the performance of our FGHC system executing on a single processor. We give the minimum time and maximum LIPS rating for various programs with the system executing on a INMOS T800 transputer and a SUN Sparc processor.

We then give measurements of the degradation in performance caused by calling the Communications Manager periodically from the toplevel. This is done by showing how the single processor performance is affected by varying the number of reductions that are made before the Communications Manager checks the message buffers.

4.6.1 Benchmark performance

Maximum performance is the LIPS rating measured when the Communications Manager is not called during program execution; the processor spends all of the execution time reducing processes.

		T800		*Sun4*	
Program	*Reductions*	*time (s)*	*LIPS*	*time (s)*	*LIPS*
nrev(150)	11478	15.1	759	0.9	12750
hanoi(15)	65537	48.5	1350	3.3	19860
fib(20)	32837	35.0	938	3.9	8385
primes(800)	22730	29.3	776	2.0	11370
queen-ls(8)	23627	45.3	521	3.1	7620

Figure 4.7: Single processor performance measurements

We have measured the performance for various benchmark programs and have listed them in figure 4.7. The source code for the benchmark programs is given in appendix A.

From these measurements we can see that the performance of the single processor configuration of the machine is of the order of 1 KLIPS on a T800 transputer and is in the order of 10 KLIPS on a SUN Sparc machine compiled under GCC-2.1. The variations in measurements arise because the interpreter does not use indexing on arguments of clauses. Instead clauses are tried sequentially from the first clause in a relation until one is reached that succeeds which means that relations with many clauses will execute more slowly on average than relations with few clauses. As an extreme example, the filter/4 relation in *queen-ls(8)* has four clauses which is significantly more than any relation in any of the other programs; hence the lower performance figures for *queen-ls(8)*.

We have to acknowledge that these performance figures are not remarkable when compared to compiler-emulator systems, which are probably at least 10 times faster. However, as an interpreter, our system gives reasonable performance figures which are comparable to the measures gained from running Prolog equivalent programs under the interpreted version of SICStus Prolog v.2.1 also compiled under GCC-2.1.

4.6.2 Communications Manager Overhead

The above measurements were taken with no Communications Manager overhead; that is, the Communications Manager was not called during execution of the program. Calling the Communications Manager, even with the single processor system, will cause some degradation in performance since at the very least the Communications Manager

will check to see if any buffers have finished receiving or finished sending. These checks will be made even in the single processor case.

We have measured this performance degradation in order to find a balance between reducing processes in the Reduction Engine and processing message buffers in the Communications Manager. To do this we made several executions of the *hanoi* benchmark program on a single T800 processor, varying the ratio between reducing goals and handling the message buffers. We used *hanoi(15)* since it does very little work during a reduction and so should represent a worst case scenario.

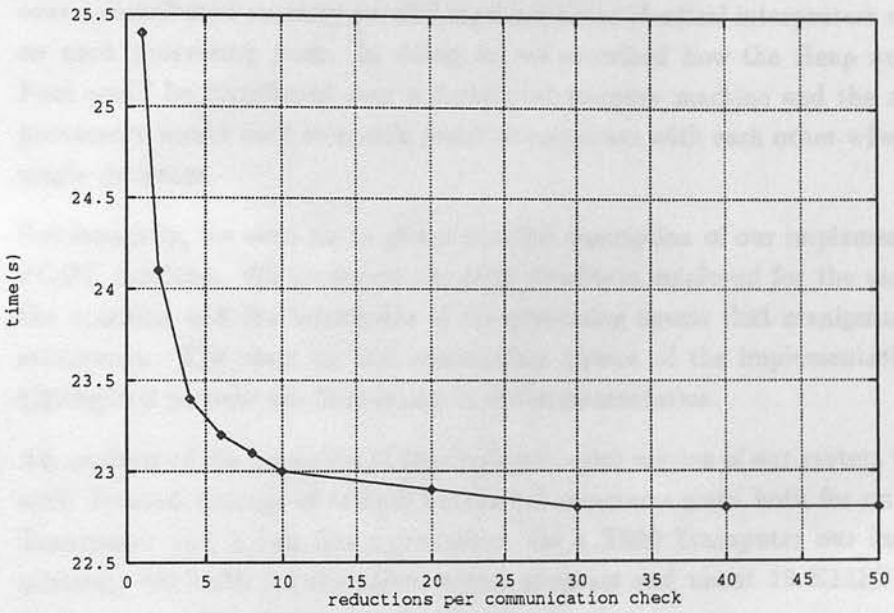

Figure 4.8: Time vs reductions per communications check

Our results are shown in the graph in figure 4.8. We show the execution time on the y-axis with the number of reductions made per message buffer check shown on the x-axis.

From the graph we can conclude that calling the Communication Manager is inexpensive in our T800/MEIKO/CSTools implementation. Even if the Communications Manager is called after every reduction there is only a $100(25.5/22.7) - 100 = 12.3\%$ maximum degredation in performance. This drops to $100(23/22.7) - 100 = 1.3\%$ at a value of 10 reductions per Communications Manager call which is a reasonable overhead to pay. As one would expect, the more reductions made between Communications Manager calls the lower the overhead will be but we must remember that the higher the number of reductions the less often communications are checked which may result in delays in processing requests on behalf of other processors. This may eventually lead to the processor starving remote processors of information and causing delays to

the overall system. A balance must be struck between minimising communications overheads and maximising the chance of dealing with messages as soon as they arrive. We have chosen the value of 10 reductions per Communications Manager call as a good compromise.

4.7 Summary

We started this chapter by outlining our overall model for implementing our FGHC interpreter. We then specialised this description into one that would suit implementation over a distributed memory parallel machine where identical interpreters would execute on each processing node. In doing so, we described how the Heap and Processing Pool could be distributed over a distributed memory machine and the messages that processors would need to enable nodes to cooperate with each other when executing a single program.

Subsequently, we went on to give a detailed description of our implementation of the FGHC machine. We presented the data structures employed for the various parts of the machine and the behaviours of the processing agents that manipulate those data structures. The start up and termination phases of the implementation were also highlighted as were the limitations of the implementation.

An analysis of the operation of the single processor version of our system was presented with detailed timings of various benchmark programs given both for our target T800 Transputer and a Sun Sparc processor. On a T800 Transputer our implementation achieves 760 LIPS for the naive reverse program and about 13 KLIPS for the same program on a Sun Sparc processor.

Finally, we concluded by analysing the inevitable performance degradation resulting from periodically switching between reducing processes and checking for incoming communications. We showed that, although the performance degradation had a reasonably low value of around 12% in the worst case of checking for messages after every reduction, the degradation could be reduced substantially without causing starvation to the system by checking for messages after every 10 reductions.

Chapter 5

Results — Interpreter

5.1 Introduction

In this chapter we present the results of the experiments carried out with our interpreter.

Each test program from *hanoi, fib, qsort, primes, queen-ls* was executed with each of the strategies *AP-NW, AP-W, NN-NW, NN-W* ten times. The experiments were repeated on differing numbers of processors, from 1 to 36, connected in a 2-D mesh topology. The results of these executions were analysed to generate performance figures using our measures.

Broadly, the purpose of these experiments was twofold: firstly, to see if there is an overall best goal distribution strategies in terms of execution time and speedup; and secondly, to see if our measures give any prediction about the performance of a program executing on a certain system configuration.

To aid readability, we use AP to refer to all-processors and NN to refer to nearest-neighbours.

5.2 Results

Here we present the results of the experiments with the interpreter system analysed according to our measures.

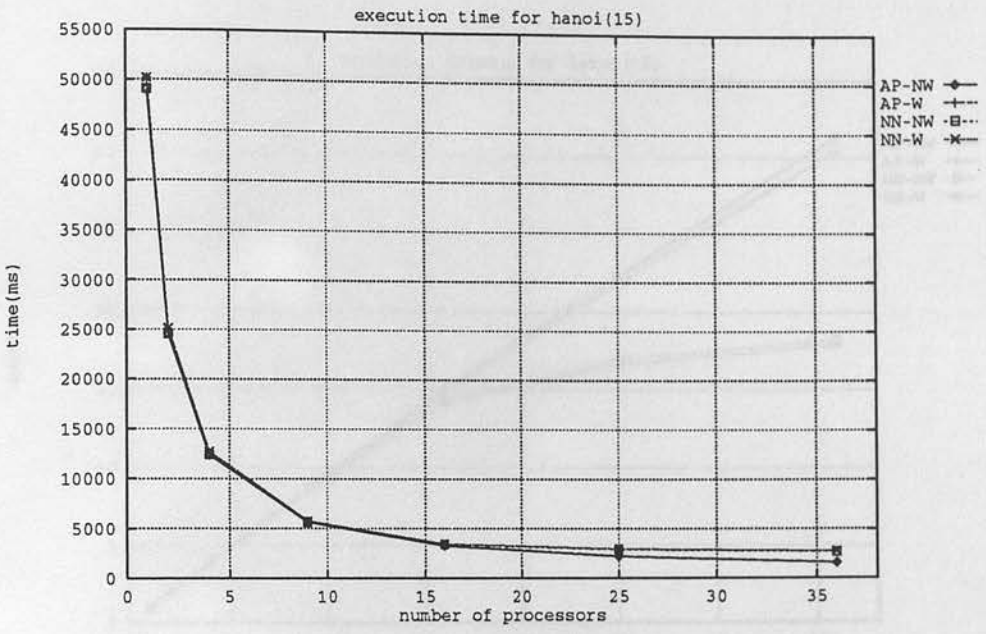

Figure 5.1: Interpreter: Execution time for *hanoi(15)*

5.2.1 hanoi

Figure 5.1 is the execution time profile for an execution of *hanoi(15)*. It shows that for small numbers of processors (1–16) the different distribution strategies execute the program in approximately the same time. For 25 and 36 processors, however, the all-processors (AP) strategies perform better than the nearest-neighbour (NN) strategies. This is better illustrated by the speedup profile of figure 5.2. This shows clearly that the AP strategies perform better than the NN strategies — at 36 processors the AP strategies show a speedup of about 30 which is about 80% of linear speedup, whereas the NN strategies show a speedup of about 18 which is about 50% of linear speedup.

Another feature to notice is that there is negligible difference in speedup between the non-weighted and weighted versions of a strategy. This is not surprising since the only goals in the *hanoi* program are *move/4* goals which all have the same weight and they execute independently. In this case, ordering by weights or not will make little difference. The weighted strategies do slightly worse than the non-weighted due to the extra complexity of the weighted algorithm — a check on weight each time a goal is queued.

The reason for the difference in performance between the AP and NN strategies can be found by analysing the load balance profile, shown in figure 5.3. The graph shows that there is a big difference in load balance when using the AP strategies and the NN

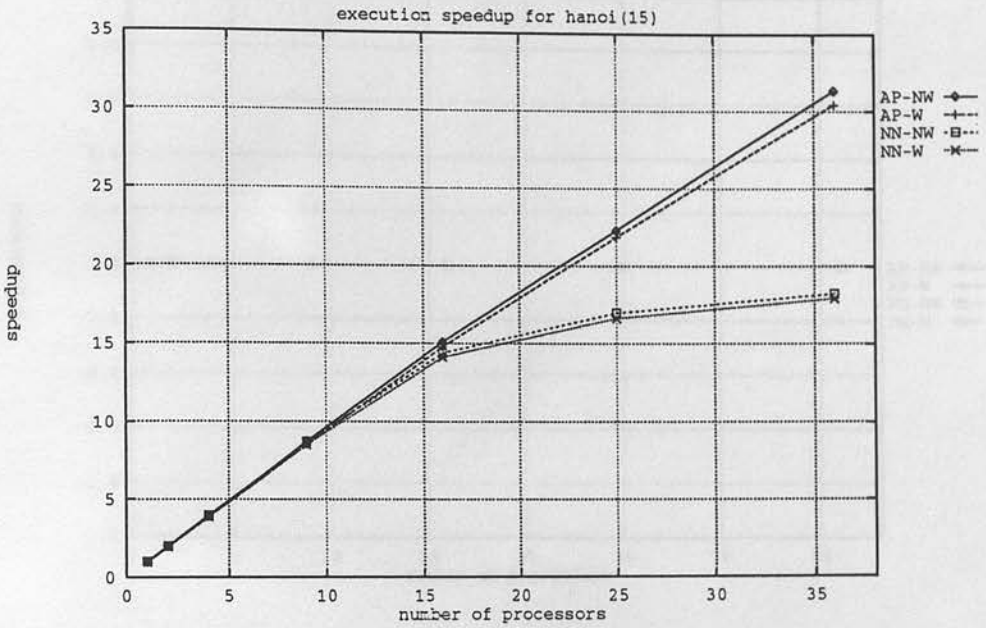

Figure 5.2: Intrepreter: Speedup for *hanoi(15)*

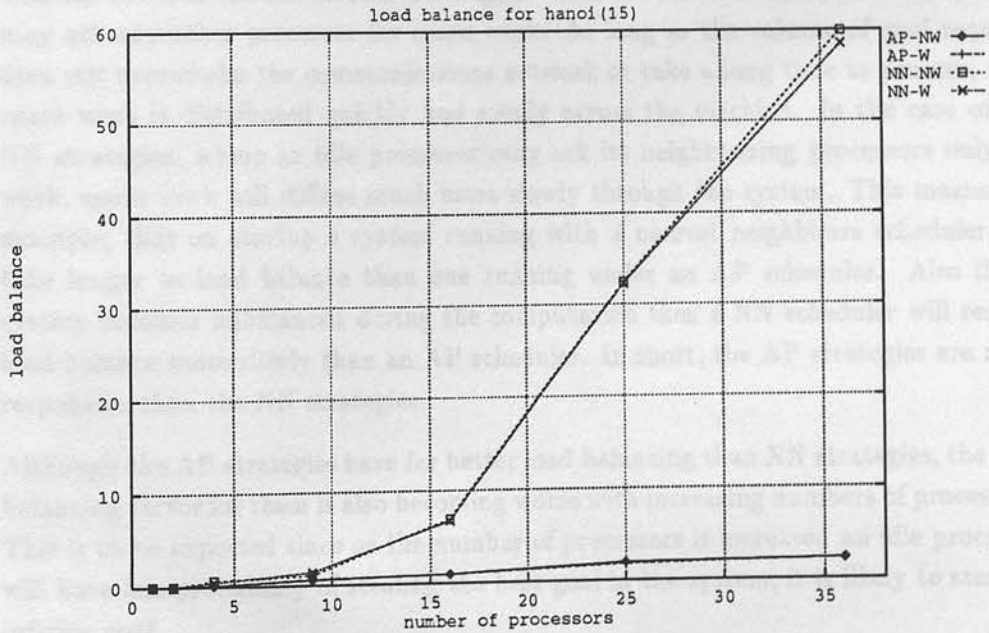

Figure 5.3: Interpreter: Load balance for *hanoi(15)*

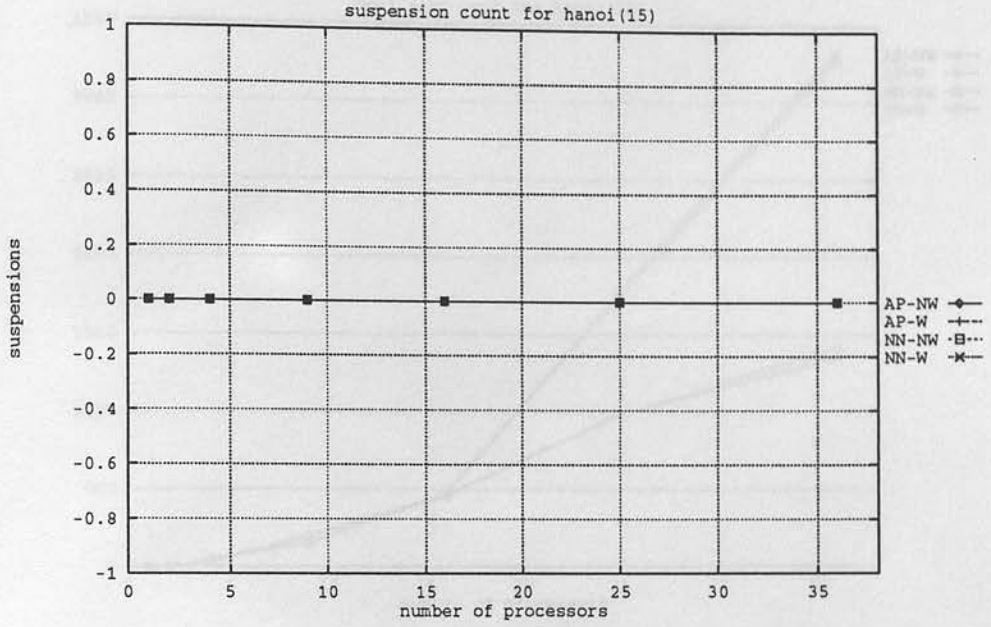

Figure 5.4: Interpreter: Suspensions for *hanoi(15)*

strategies; the AP strategies show extremely good load balance, but the NN strategies show significantly worse load balance. This is not surprising when we examine the differences between the NN and AP strategies. With the AP strategies, an idle processor may ask any other processor for spare work. As long as the volume of goal requests does not overwhelm the communications network or take a long time to process, then spare work is distributed quickly and evenly across the machine. In the case of the NN strategies, where an idle processor may ask its neighbouring processors only for work, spare work will diffuse much more slowly through the system. This means, for example, that on startup a system running with a nearest neighbours scheduler will take longer to load balance than one running under an AP scheduler. Also if the system becomes unbalanced during the computation then a NN scheduler will restore load balance more slowly than an AP scheduler. In short, the AP strategies are more responsive than the NN strategies.

Although the AP strategies have far better load balancing than NN strategies, the load balancing factor for them is also becoming worse with increasing numbers of processors. This is to be expected since as the number of processors is increased an idle processor will have less probability of stealing the best goal in the system; it is likely to steal an inferior goal.

Because there are no dependencies between the goals in an execution of the *hanoi* program, there should be no suspensions or binding requests, and this is confirmed by

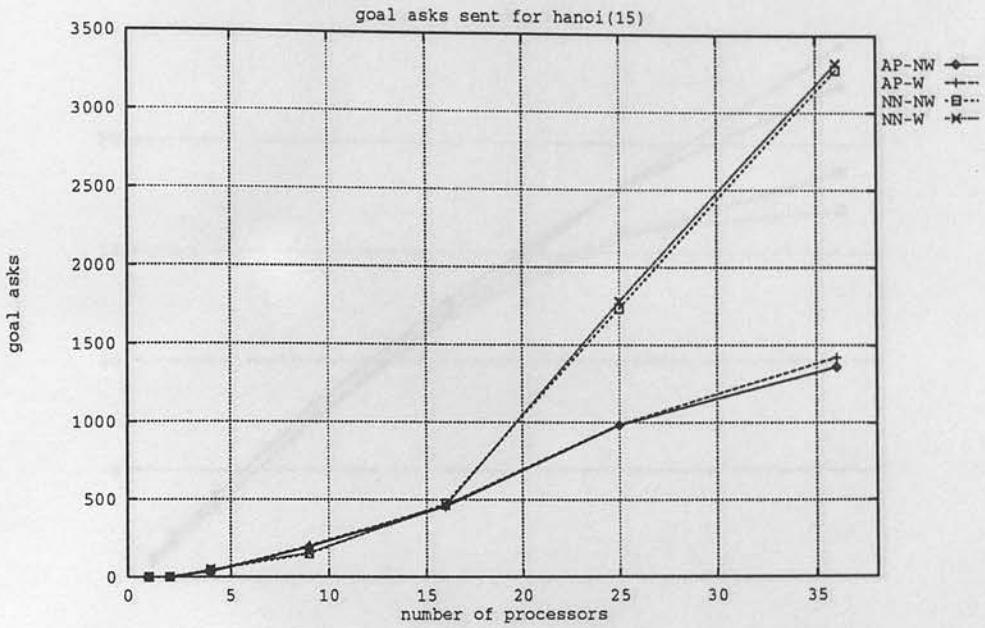

Figure 5.5: Interpreter: Goal requests for *hanoi(15)*

plotting the graph for suspensions shown in figure 5.4. The same graph also serves for the binding requests profile.

Figure 5.5 is a graph of goal requests against processors. This shows that the NN strategies send substantially more goal requests than the AP strategies. This may be an indicator of the idleness of processors. Processors that have no work send goal requests and goal distribution strategies that cause many processors to be idle should cause many goal requests. This certainly seems to be the case for the *hanoi* program.

To summarise what we have learnt from the execution of the *hanoi* program:

- The AP strategies give very good speedups.
- The NN strategies give much worse speedup than the AP strategies.
- Load balancing is an important factor. The AP strategies perform better than the NN strategies because they have superior load balancing properties.
- If goals are ordered according to weight then there is a small performance degradation due to the increased complexity of the algorithm when compared with the non-weighted strategy.
- The number of goal requests made may also be an indicator of the effectiveness of a distribution strategy. The AP strategies generate substantially less goal requests than the NN strategies.

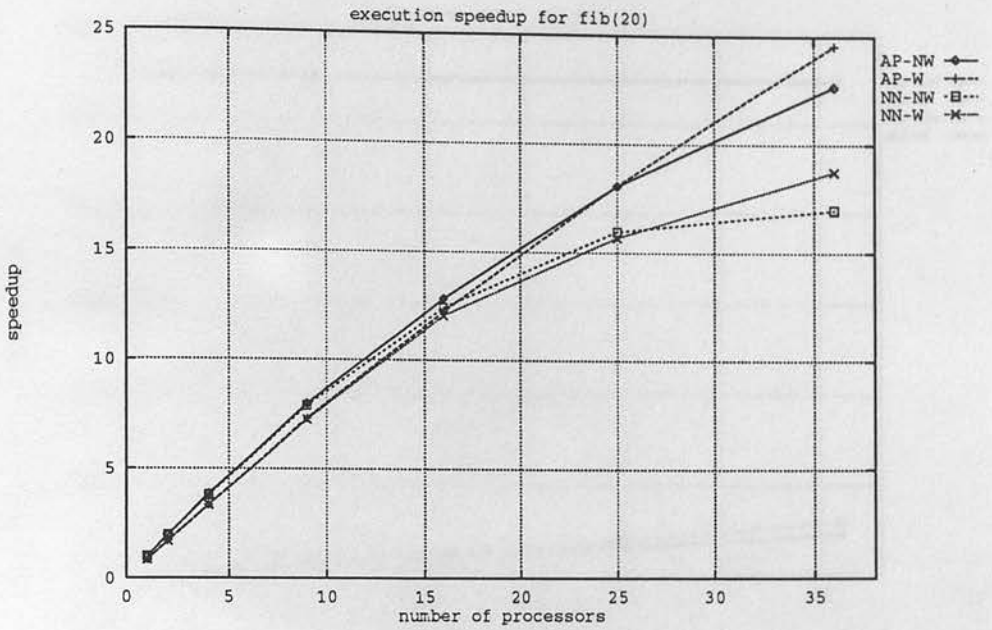

Figure 5.6: Interpreter: Speedup for *fib(20)*

5.2.2 fib

The speedup graph for the *fib* program is shown in figure 5.6. For small numbers of processors (1–16) there is not a big difference between the strategies, but there is a discernible grouping: the weighted strategies perform slightly worse than the non-weighted strategies.

For larger numbers of processors the groupings have changed and the differences between the strategies are more pronounced. Now the AP strategies are performing better than the NN strategies, although at 36 processors the weighted version of a strategy performs significantly better than the equivalent non-weighted strategy.

The suspensions profile, shown in figure 5.7, provides part of the explanation for the performance differences between the strategies. It shows clearly that the weighted strategies perform a large number of suspensions (11,000), whereas the non-weighted strategies perform significantly fewer (less than 1000). The suspensions for the weighted strategies are constant regardless of the number of processors used.

This behaviour is reasonable when we consider the differences in the way the *fib* program (see figure 5.8) executes with the different strategies. With the weighted strategies the *add/3* goal has weight 0 but the *fib/2* goals spawned with it have weight 2, which means that the *add/3* will be executed first each time. But the *add/3* goal is dependent on the *fib/3* goals for its input values, so when it is executed it will immediately sus-

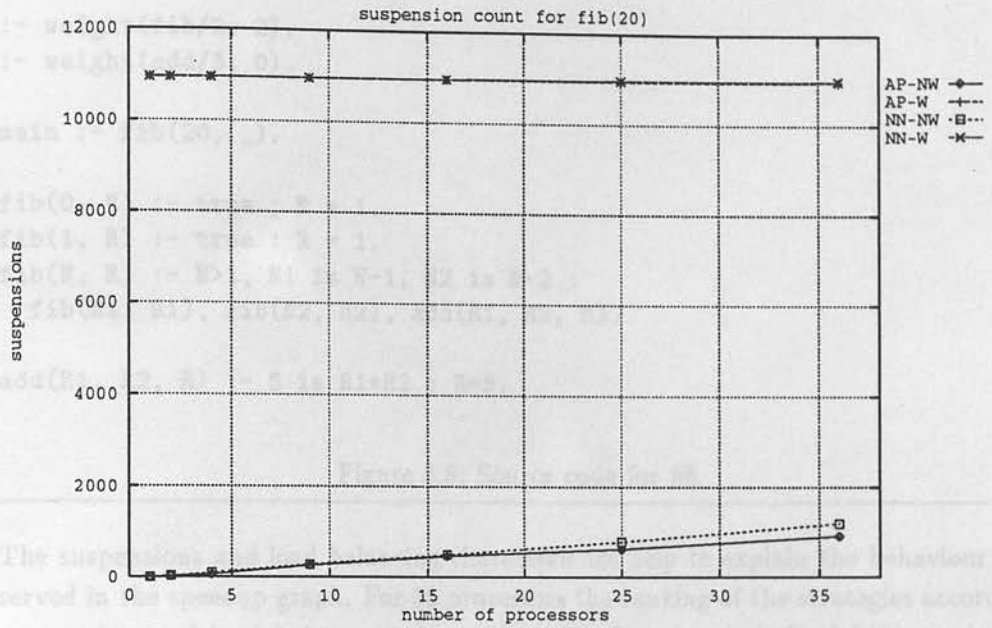

Figure 5.7: Interpreter: Suspensions for *fib(20)*

pend. In this way every single *add/3* goal will suspend locally which accounts for the very high constant suspension rate for the weighted strategies. Since the *add/3* goals suspend early, it will only be *fib/2* goals that will be distributed to remote processors. In this mode the execution of the *fib* program is similar to the *hanoi* program except with an extra suspension per reduction of *fib/2*.

In contrast, for the non-weighted strategies, the *add/3* goal is ordered after the *fib/2* goals in the program, so it is the *add/3* goals that will be distributed remotely first. They are then likely to suspend on the remote processor since the values of both the *fib/2* goals are unlikely to have been computed; the remote processor will then steal another goal which may be a *fib/2* goal or an *add/3* goal. But basically, the suspensions are caused by *add/3* goals suspending on remote variables.

Figure 5.9 is the load balance plot for *fib*. This shows that the AP-W strategy has the best load balance characteristic followed by AP-NW. The NN strategies have significantly worse load balancing performance, but with NN-W better than NN-NW. The AP strategies are better at load balancing than the NN strategies but the weighted versions of strategies are better than the non-weighted versions. Notice that with small numbers of processors (< 16) there is little difference between the three best strategies (AP-W, AP-NW, NN-W) but that NN-NW performs worse than the others over the whole range of processors. For larger numbers of processors the differences between the strategies become obvious.

```
:- weight(fib/2, 2).
:- weight(add/3, 0).

main :- fib(20, _).

fib(0, R) :- true : R = 1.
fib(1, R) :- true : R = 1.
fib(N, R) :- N>1, N1 is N-1, N2 is N-2 :
  fib(N1, R1), fib(N2, R2), add(R1, R2, R).

add(R1, R2, R) :- S is R1+R2 : R=S.
```

Figure 5.8: Source code for *fib*.

The suspensions and load balancing characteristics help to explain the behaviour observed in the speedup graph. For 36 processors the ranking of the strategies according to speedup and load balance is the same, indicating that it is load balance that is significant for large numbers of processors.

With small numbers of processors, when the load balance characteristics of the strategies are similar, it is the number of suspensions that correlates with the ranking of strategies by speedup.

The question we would like to answer is: *Why do the weighted strategies perform better than the non-weighted for large numbers of processors?* We have seen that they have better load balancing characteristics but why is that so? Whatever the factor is, it must be large enough to counter the effect of the large numbers of suspensions which the weighted strategies incur.

The answer may be that, when using the non-weighted strategies, idle processors are stealing mainly *add/3* goals. The overhead of sending these goals is large compared to the amount of execution they provide, and they are likely to suspend straight away. These factors may cause a lot of wasted effort in message sending and suspension overheads. With the weighted strategies there is an overhead in that the *add/3* goals immediately suspend on execution, causing a lot of local suspensions, but goals which are stolen are *fib/2* goals which can execute immediately and potentially involve a reasonable amount of work. The time spent performing the local suspensions, however, may be less than the inefficiency introduced by sending out *add/3* goals to immediately suspend as happened in the non-weighted strategies. Hence the weighted strategies are better than the non-weighted strategies for large number of processors. This is an instance of the Tick weights algorithm working.

Figure 5.10 is a graph of the number of binding requests for *fib*. The number of binding

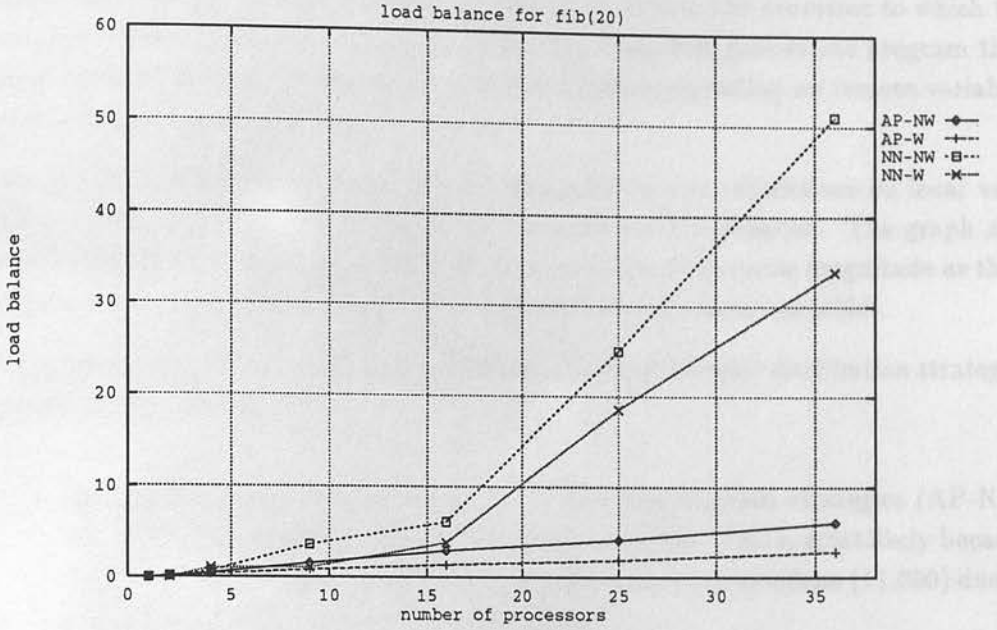

Figure 5.9: Interpreter: Load balance for *fib(20)*

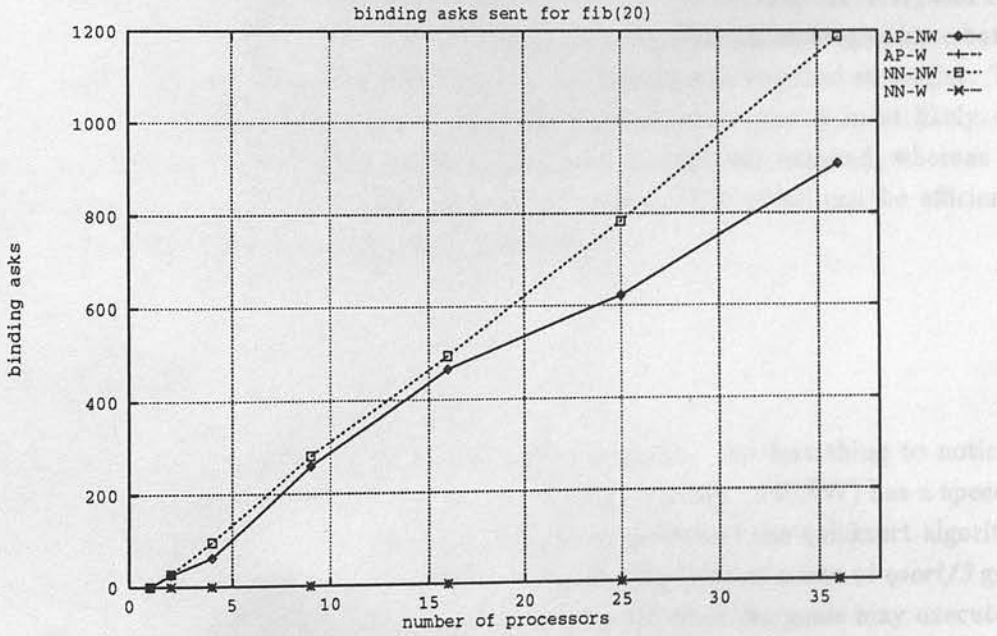

Figure 5.10: Interpreter: Binding requests for *fib(20)*

requests is related to the number of goals that suspend on remote variables since such a suspension is likely to cause a binding request to be sent to the processor to which the variable refers[1]. Since the *add/3* goal is the only dependent goal in the program then these remote suspensions must be due to *add/3* goals suspending on remote variables after being stolen by an idle processor.

The graph shows that the weighted strategies generate no suspensions on local variables, which confirms our analysis of the execution of the program. The graph also shows that the AP strategies generate binding requests of the same magnitude as their suspensions, which confirms that the suspensions are on remote variables.

We summarise the factors affecting the characteristics of the goal distribution strategies executing the *fib* program:

- For small numbers of processors, (1–16) the non-weighted strategies (AP-NW, NN-NW) perform better than the weighted strategies. This is most likely because the weighted strategies incur a large number of local suspensions (11,000) due to *add/3* goals being ordered before *fib/2* goals.
- For large numbers of processors, the AP strategies perform better than the NN strategies. This is because of their superior load balancing characteristics.
- For large numbers of processors, the weighted version of a strategy performs better than the non-weighted version: AP-W does better than AP-NW, and NN-W does better than NN-NW. This is because the weighted strategies have better load balancing characteristics than their equivalent non-weighted strategies. The worse load balancing effects of the non-weighted strategies is most likely due to idle processors stealing *add/3* goals which immediately suspend, whereas the same non-weighted strategy will always steal a *fib/2* which can be efficiently executed (apart from the local suspensions).

5.2.3 qsort

Figure 5.11 is the speedup graph for the *qsort* program. The first thing to notice is that the speedup values are not very good: the best strategy (AP-NW) has a speedup factor of 6 at 36 processors. This is because of the nature of the quicksort algorithm (see figure 5.12). Parallel execution of the program will result in a tree of *qsort/3* goals dependent on a *part/4* goal for input data. Although all of the goals may execute in

[1]This is complicated by the Remote Binding Array. If two goals suspend on the same remote variable then a binding request will only be sent for the first one. Therefore, the number of binding request is only an indicator of the number of suspensions on remote variables.

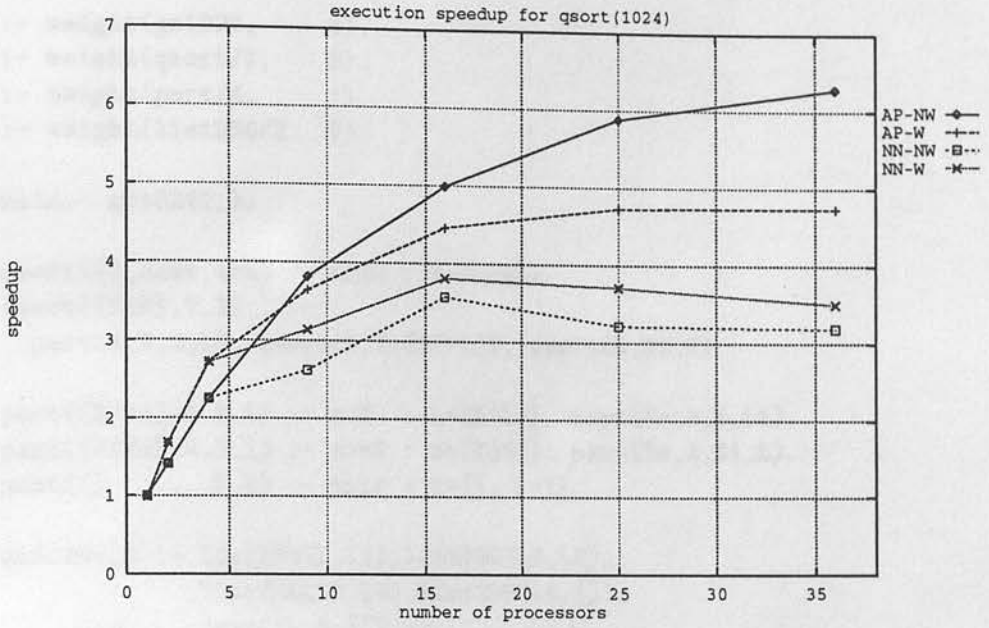

Figure 5.11: Interpreter: Speedup for *qsort(1024)*

stream AND-parallel, in the worst case the partition goal will produce values on its two output streams at half the rate that it processes data on its input stream. In other words, the parallelism of the *qsort* program is bounded by the *part/4* goals.

Looking at the relative performance of the goal distribution strategies, it is clear that AP-NW is the best strategy with a speedup factor of over 6 for 36 processors. It is followed by AP-W which has peaked at a speedup factor of just less than 5 at 25 processors: using more than 25 processors would be a waste of resources.

The NN strategies perform significantly worse than the AP strategies. They both peak with a speedup of between 3.5 and 4 for 16 processors. With more than 16 processors, speedup declines.

For the AP strategies the non-weighted strategy (AP-NW) performs significantly better than the weighted strategy (AP-W). For the NN strategies the opposite is true: NN-W performs better than NN-NW.

The same pattern is repeated in the load balancing graph show in figure 5.13 which suggests (again) that load balancing is a significant factor affecting performance.

Initially it is difficult to see why adding weights to a strategy should make any difference to the execution of the *qsort* program. The *part/4* goal has a lower weight than the *qsort/3* goal and so will be scheduled locally before them, and also in the non-weighted program the *part/4* is ordered before the *qsort/3* goals.

```
:- weight(go1024,     3).
:- weight(qsort/3,    3).
:- weight(part/4,     1).
:- weight(list256/2,  0).

main:- go1024(_).

qsort([],Rest,Ans) :- true : Rest=Ans.
qsort([X|R],Y,T)   :-
  part(R,X,S,L), qsort(S,Y,[X|Y1]), qsort(L,Y1,T).

part([X|Xs],A,S,L) :- A<X  : L=[X|L1], part(Xs,A,S,L1).
part([X|Xs],A,S,L) :- A>=X : S=[X|S1], part(Xs,A,S1,L).
part([]    ,_,S,L) :- true : S=[], L=[].

go1024(_) :- list256(L,L2),list256(L2,L3),
             list256(L3,L4),list256(L4,[]),
             qsort(L,A,[]).
```

Figure 5.12: Source code for *qsort*.

The behaviour of the program is more complex than that, however. Consider the scenario where processor A reduced a *qsort/3* goal to a *part/4* goal and two *qsort/3* goals. While processor A is processing the first *part/4* goal, processor B steals one of the *qsort/3* goals which then reduces to a *part/4* and two *qsort/3*s. The *part/4* will reduce for a while but may then suspend waiting on more input from processor A, and so it sends a binding request and suspends the *part/4*. All its other goals have been stolen in the meantime, so it steals another *qsort* from another process, but it also suspends waiting for input and send a binding request. Let us say that the bindings sent as a result of the requests arrive at processor B at more or less the same time. Now the *part/4* goal and the *qsort/3* goal can be rescheduled. If using the weighted strategy then the *part/4* goal will be scheduled before the *qsort/3*. When using a non-weighted strategy, the order in which the binding messages arrive affects the goal ordering. This shows that it is possible to get different goal orderings with the different strategies, although it is far from clear whether this explains the performance effects we observe.

Figure 5.14 is the suspensions plot for *qsort*. It shows that the weighted strategies have more suspensions than the non-weighted strategies, and that NN-NW has significantly fewer suspensions than the other three strategies; but NN-NW also has the worst speedup characteristic.

Figure 5.15 is the binding request plot for *qsort*. It shows that the AP-NW strategy has by far the most binding requests, the other three strategies generating half as many

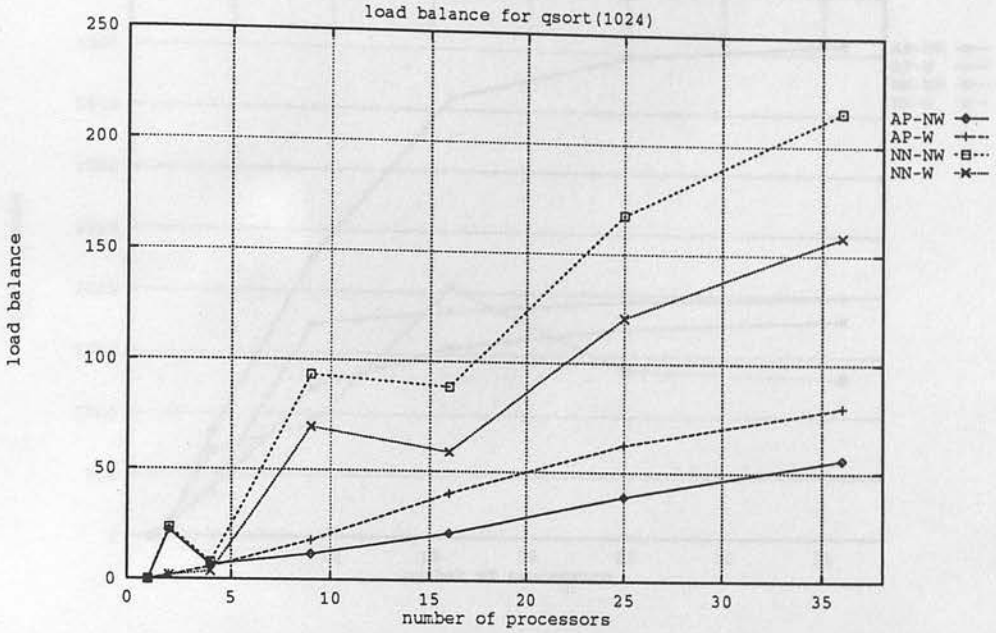

Figure 5.13: Interpreter: Load balance for *qsort(1024)*

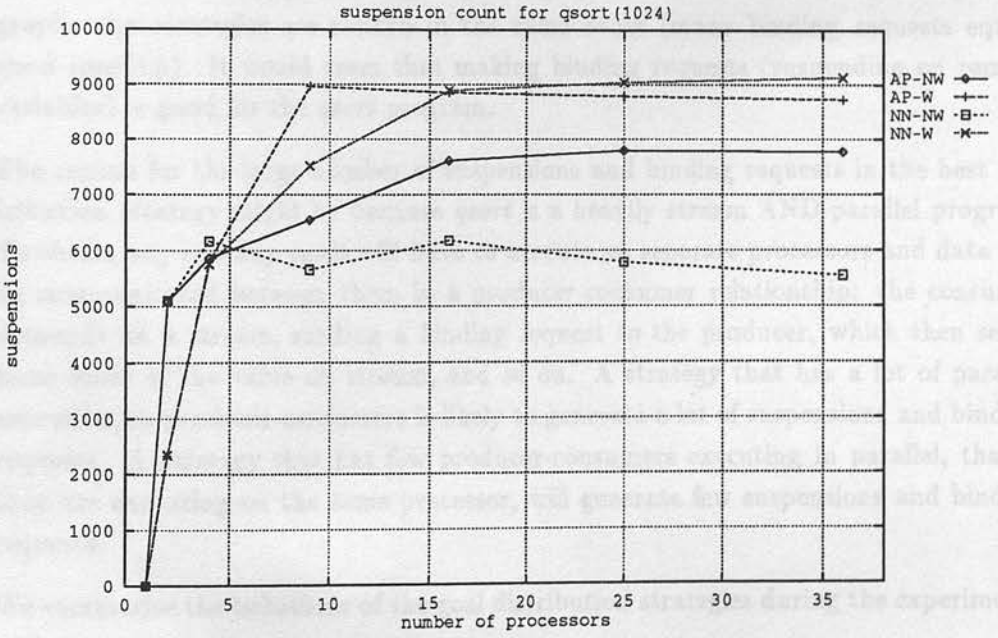

Figure 5.14: Interpreter: Suspensions for qsort(1024)

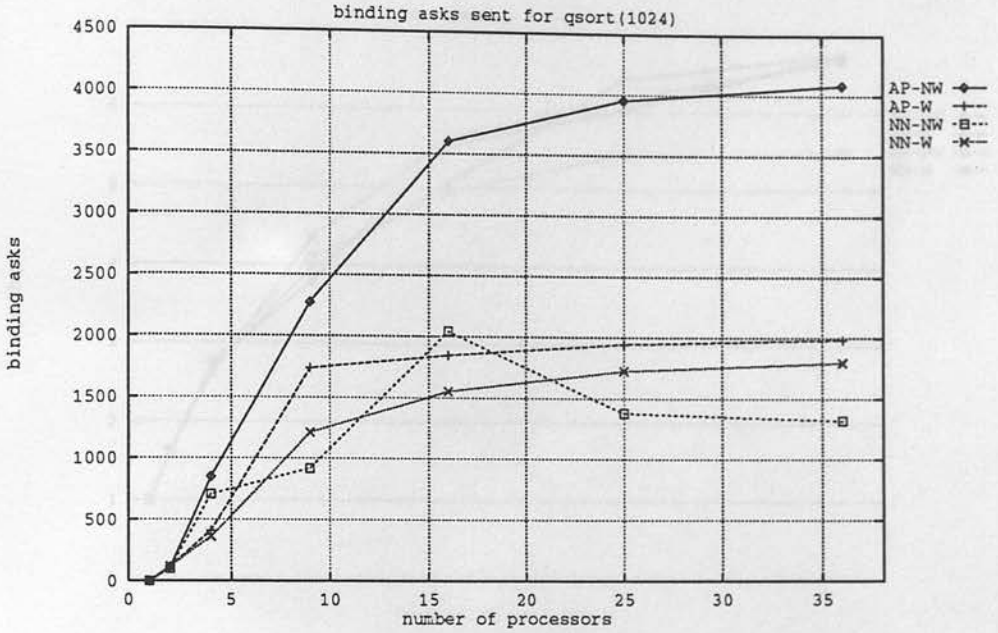

Figure 5.15: Interpreter: Binding requests for *qsort(1024)*

for large numbers of processors. Also, for AP-NW the number of binding requests is similar to the number of suspensions indicating that the majority of suspensions are on remote variables. The binding requests graph has a similar form to the speedup graph: the strategies are ranked in the same order (many binding requests equals good speedup). It would seem that making binding requests (suspending on remote variables) is good for the *qsort* program.

The reason for the large number of suspensions and binding requests in the best distribution strategy might be because *qsort* is a heavily stream AND-parallel program. To obtain any speedup goals will have to execute on separate processors and data will be communicated between them in a producer-consumer relationship: the consumer suspends on a stream, sending a binding request to the producer, which then sends some more of the value on stream, and so on. A strategy that has a lot of parallel execution on producer-consumers is likely to generate a lot of suspensions and binding requests. A strategy that has few producer-consumers executing in parallel, that is they are executing on the same processor, will generate few suspensions and binding requests.

We summarise the behaviour of the goal distribution strategies during the experiments with *qsort*:

- The AP strategies are better than the NN strategies, with AP-NW performing

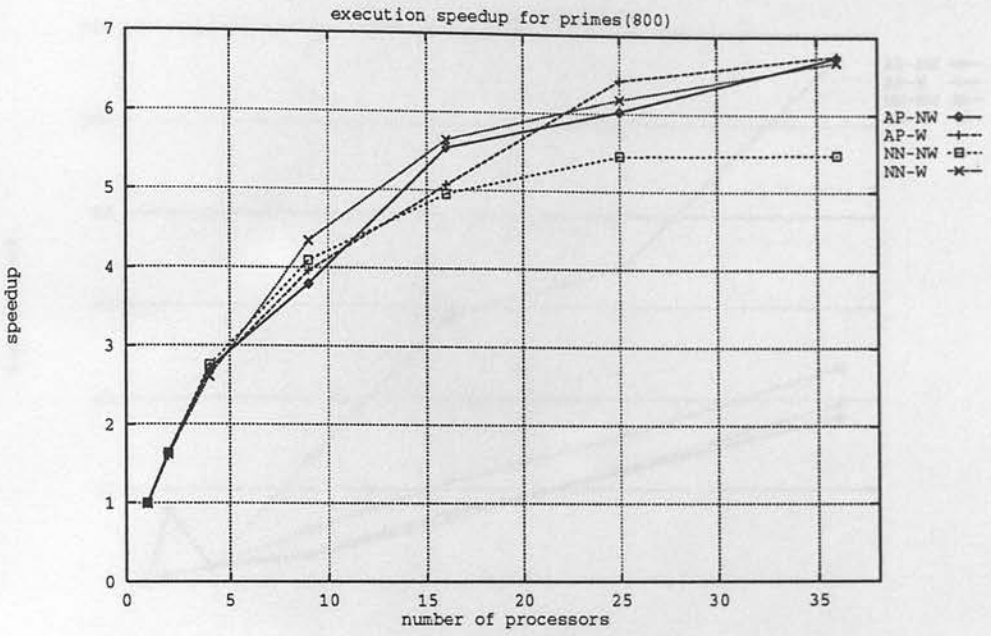

Figure 5.16: Interpreter: Speedup for *primes*

the best.

- Similarly, the AP strategies have better load balance characteristics than the NN strategies, which again suggests that good load balancing is an indication of good performance.

- For the AP strategies using weights is bad, but for the NN strategies using weights is good principally through better load balance. It is not clear why NN-W has a better load balancing characteristic than NN-NW. The only indicator is that the suspending on remote variables may also be an indicator of good performance and that NN-W produces more binding requests than NN-NW. This may be because a program that is stream AND-parallel as *qsort* is has to obtain speedup by having producer and consumer goals executing on different processors: communicating between the goals generates a lot of suspensions on remote variables which is reflected in the binding request count.

5.2.4 primes

Figure 5.16 is the speedup plot for the experiments with *primes*. The first thing to note is that the speedup factor is low for all of the strategies: the best speedup is about 7 for 36 processors. This is not surprising behaviour from the *primes* program since its

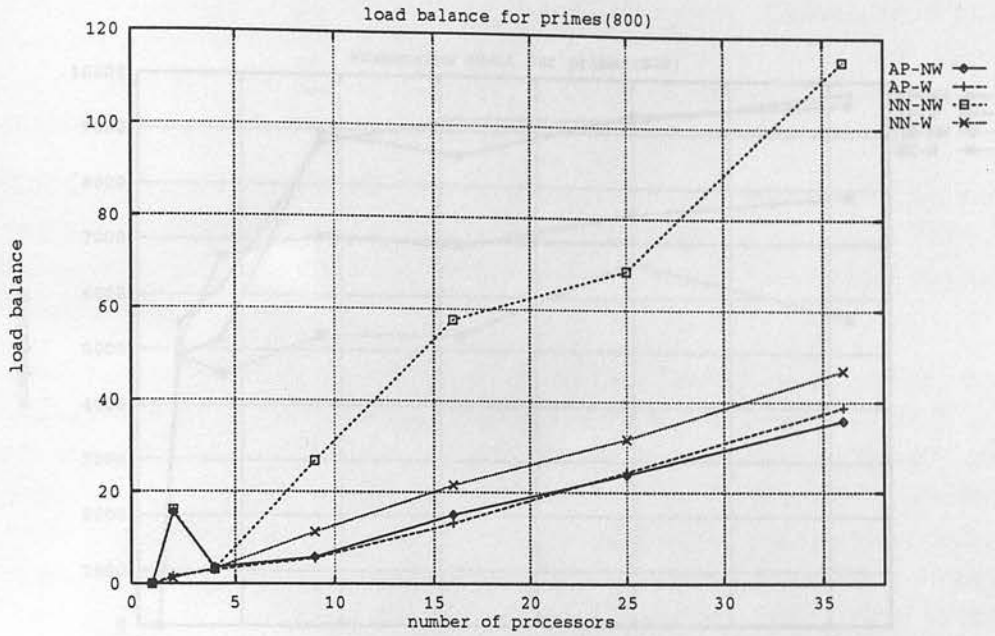

Figure 5.17: Interpreter: Load balance for *primes(800)*.

execution creates a pipeline of filter processes, each filtering out multiples of a prime from a stream of integers, and the stream of integers becomes sparser the further away the filter process is from the integer generator; a large number of the filter processes will be suspended a lot of the time which will limit the parallelism available to be exploited.

The speedup graph shows that for large numbers of processors (25 and 36) there is no significant difference between the top three distribution strategies (AP-NW, AP-W, NN-W), but that NN-NW performs worse than the others.

The load balance graph, figure 5.17, clearly shows that NN-NW has very much worse load balancing characteristics than the other three strategies which have similar load balancing characteristics. This shows that using weights with the NN strategies improves load balancing. Employing weights in the AP case seems to have no effect on load balancing. These findings (again) compare strongly with our observations of speedup.

Figure 5.18 is a plot of the average number of suspensions for *primes*. This shows that the AP strategies generate a lot of suspensions: for 9 processors or more the suspension count has reached 9000. The NN strategies generate fewer suspensions with NN-NW generating the least number of suspensions. However, NN-NW also shows the worst speedup characteristic.

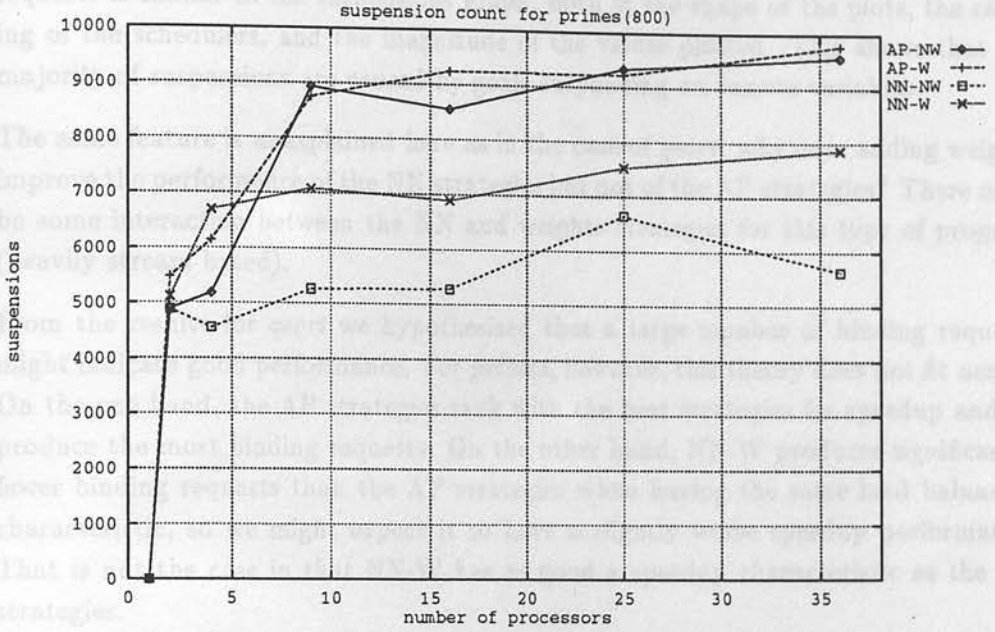

Figure 5.18: Interpreter: Suspensions for *primes(800)*.

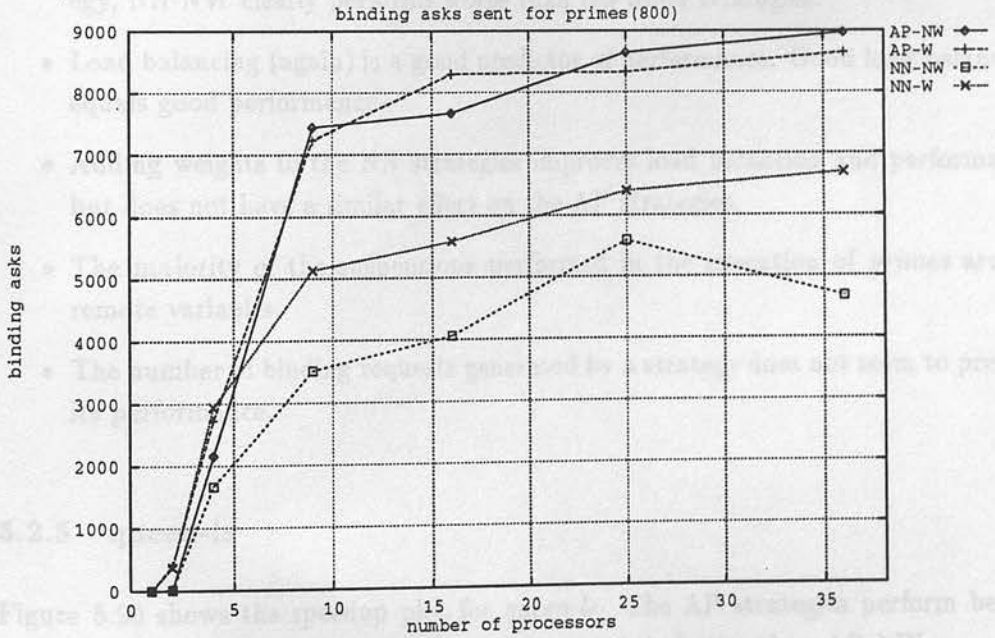

Figure 5.19: Interpreter: Binding requests for *primes(800)*.

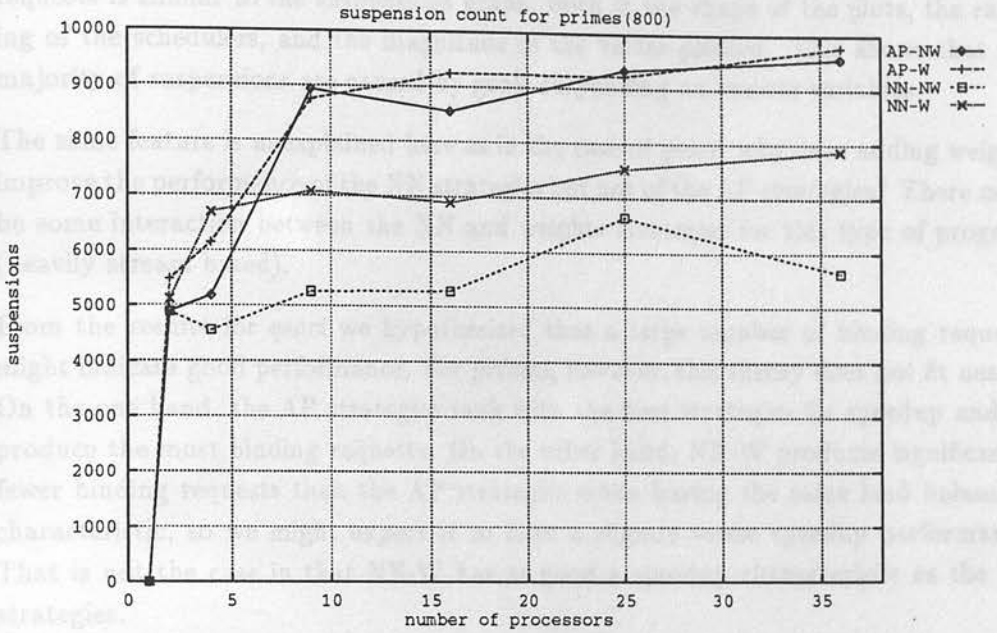

Figure 5.18: Interpreter: Suspensions for *primes(800)*.

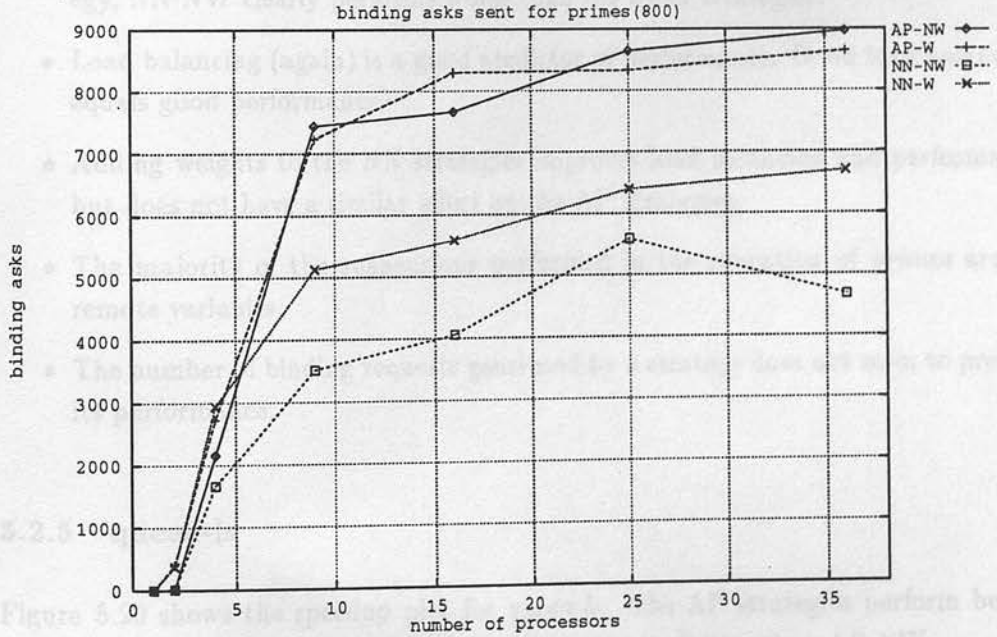

Figure 5.19: Interpreter: Binding requests for *primes(800)*.

The graph in figure 5.19 shows binding requests for *primes*. The number of binding requests is similar to the suspensions graph, both in the shape of the plots, the ranking of the schedulers, and the magnitude of the values plotted. This shows that the majority of suspensions are caused by goals suspending on remote variables.

The same feature is unexplained here as in the case of *qsort*: why does adding weights improve the performance of the NN strategies but not of the AP strategies? There must be some interaction between the NN and weights strategies for this type of program (heavily stream based).

From the results for *qsort* we hypothesised that a large number of binding requests might indicate good performance. For *primes*, however, this theory does not fit neatly. On the one hand, the AP strategies rank with the best strategies for speedup and do produce the most binding requests. On the other hand, NN-W produces significantly fewer binding requests than the AP strategies while having the same load balancing characteristic, so we might expect it to have a slightly worse speedup performance. That is not the case in that NN-W has as good a speedup characteristic as the AP strategies.

We summarise the behaviour of the goal distribution strategies during the experiments with *primes*:

- Three strategies perform equally well: AP-NW, AP-W, NN-W. The other strategy, NN-NW clearly performs worse than the other strategies.
- Load balancing (again) is a good predictor of performance. Good load balancing equals good performance.
- Adding weights to the NN strategies improves load balancing and performance but does not have a similar effect on the AP strategies.
- The majority of the suspensions performed in the execution of *primes* are on remote variables.
- The number of binding requests generated by a strategy does not seem to predict its performance.

5.2.5 queen-ls

Figure 5.20 shows the speedup plot for *queen-ls*. The AP strategies perform better than the NN strategies with AP-W performing slightly better than AP-NW.

Figure 5.21 shows the load balance plot for *queen-ls*. The NN strategies have significantly worse load balancing characteristics than the AP strategies. NN-NW is better at

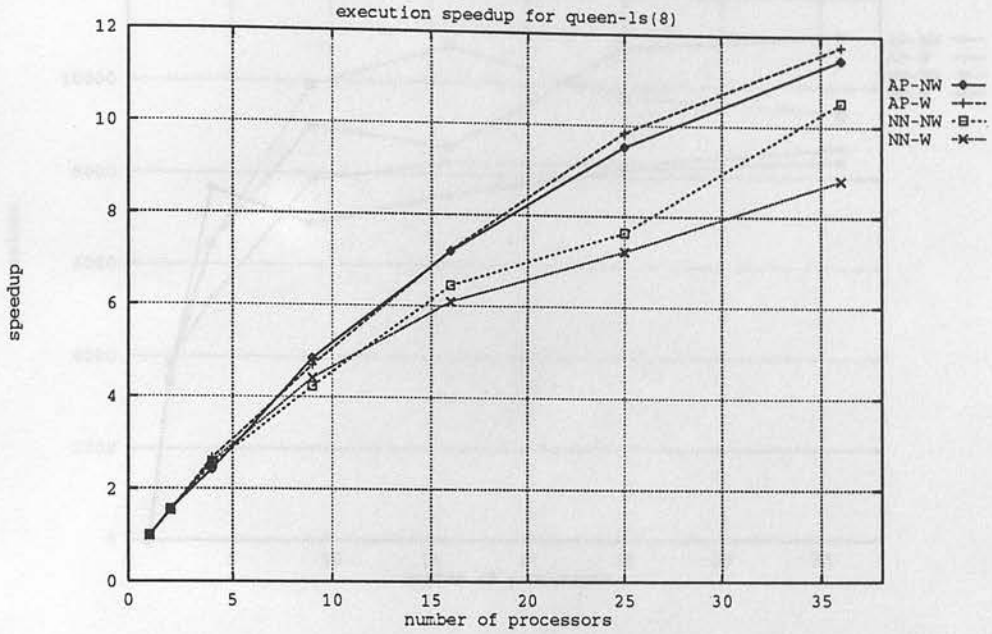

Figure 5.20: Interpreter: Speedup for *queen-ls(8)*

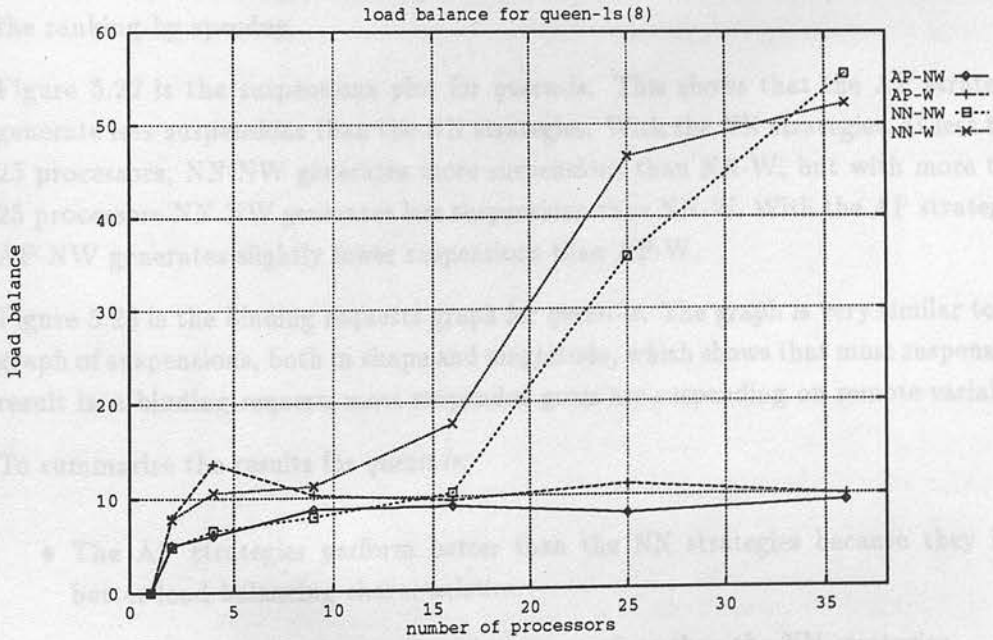

Figure 5.21: Interpreter: Load balance for *queen-ls(8)*

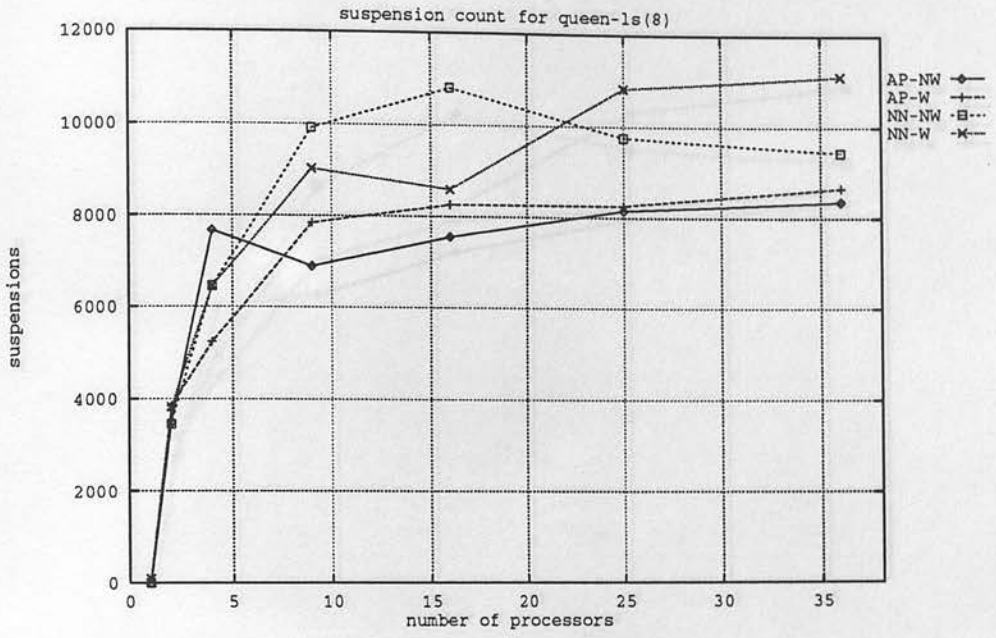

Figure 5.22: Interpreter: Suspensions for *queen-ls(8)*

load balancing than NN-W which is likely to be the main reason why NN-NW performs better than NN-W on speedup. There is little difference between AP-NW and AP-W in load balance although AP-NW performs slightly better than AP-W, the opposite to the ranking by speedup.

Figure 5.22 is the suspensions plot for *queen-ls*. This shows that the AP strategies generate less suspensions than the NN strategies. With the NN strategies, at less than 25 processors, NN-NW generates more suspensions than NN-W, but with more than 25 processors NN-NW generates less suspensions than NN-W. With the AP strategies, AP-NW generates slightly fewer suspensions than AP-W.

Figure 5.23 is the binding requests graph for *queen-ls*. The graph is very similar to the graph of suspensions, both in shape and magnitude, which shows that most suspensions result in a binding request; most suspended goals are suspending on remote variables.

To summarise the results for *queen-ls*:

- The AP strategies perform better than the NN strategies because they have better load balancing characteristics.
- The AP strategies also generate less suspensions than the NN strategies.
- For all the strategies the suspensions counts and binding request counts are similar showing that most suspensions are on remote variables.

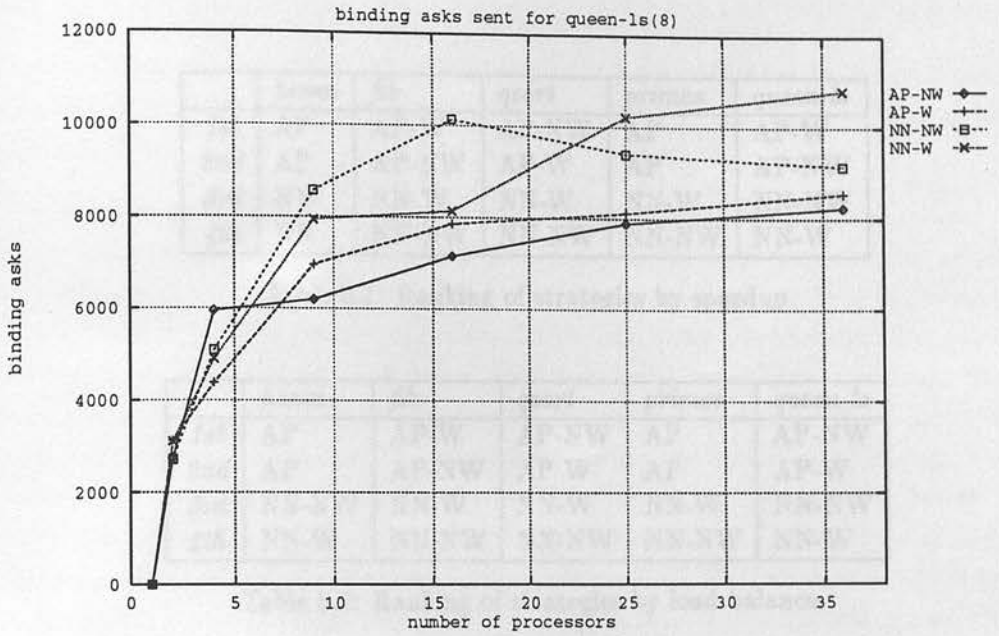

Figure 5.23: Interpreter: Binding requests for *queen-ls(8)*

5.3 Discussion

As an overview of the experiments, tables 5.1–5.5 show the rankings of goal distribution strategies against program and measure. Where ranking of the strategies could not be made there is an entry of 'n/r', meaning 'no ranking'. Similarly, an entry of 'AP' means that the AP strategies could not be ranked (AP-W and AP-NW are given the same ranking), and an entry of '-NW' means that the non-weighted strategies could not be ranked (AP-NW and NN-NW are given the same rank); the same convention applies to entries of 'NN' and '-W'.

We will now discuss the results of the experiments and draw some general conclusions about them.

Across all the programs, the AP strategies have better load balancing characteristics than the NN strategies (see table 5.2). This is because in the NN strategy idle processors may only ask nearest neighbours for work and so work diffuses across the processors less quickly than in the case of the AP strategies. With four processors or less there should be no difference between the AP and NN strategies since all processors are nearest neighbours and this is observed in all the results. With more than four processors the differences between the AP and NN strategies should become more pronounced as the proportion of processors that are neighbours to each other diminishes[2]

[2]With four processors the proportion of neighbours to all processors is 100%, but with 25 processors

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	AP	AP-W	AP-NW	AP	AP-W
2nd	AP	AP-NW	AP-W	AP	AP-NW
3rd	NN	NN-W	NN-W	NN-W	NN-NW
4th	NN	NN-NW	NN-NW	NN-NW	NN-W

Table 5.1: Ranking of strategies by speedup

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	AP	AP-W	AP-NW	AP	AP-NW
2nd	AP	AP-NW	AP-W	AP	AP-W
3rd	NN-NW	NN-W	NN-W	NN-W	NN-NW
4th	NN-W	NN-NW	NN-NW	NN-NW	NN-W

Table 5.2: Ranking of strategies by load balance

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	n/r	-NW	NN-NW	NN-NW	AP-NW
2nd	n/r	-NW	AP-NW	NN-W	AP-W
3rd	n/r	-W	AP-W	AP	NN-NW
4th	n/r	-W	NN-W	AP	NN-W

Table 5.3: Ranking of strategies by suspensions

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	n/r	-W	NN-NW	NN-NW	AP-NW
2nd	n/r	-W	NN-W	NN-W	AP-W
3rd	n/r	-NW	AP-W	AP	NN-NW
4th	n/r	-NW	AP-NW	AP	NN-W

Table 5.4: Ranking of strategies by binding requests

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	AP	NN-W	NN-W	NN-W	NN-NW
2nd	AP	AP-W	NN-NW	NN-NW	NN-W
3rd	NN	NN-NW	AP-NW	AP	AP-W
4th	NN	AP-NW	AP-W	AP	AP-NW

Table 5.5: Ranking of strategies by goal requests

and again this is supported by our observations.

Load balancing has a dominant effect on performance such that in many cases the strategy that has the best load balancing characteristic also has the best speedup characteristic.

The number and type of suspensions can also affect program execution. This was most noticeable in the execution of the *fib* program. When executing *fib*, the weighted strategies produce many local suspensions, and so with small numbers of processors, where there is little distinction between distribution strategies through load balance, the weighted strategies do not perform as well as the non-weighted strategies. But as the number of processors is increased, load balancing becomes more dominant than local suspensions, and so the AP-W strategy, having the best load balancing characteristic has the best speedup performance. The weighted strategies have better load balancing characteristics than the non-weighted because, with the weighted strategies, the dependent *add/3* goal suspends early locally leaving *fib/2* goals to execute in independent AND-parallelism. With the non-weighted strategies, the *add/3* goal will be sent out to a remote processor but will suspend on a remote variable immediately and the processor will then have to steal another goal from another processor.

We can deduce that it is a good general rule to suspend dependent goals locally early rather than allow them to be stolen to suspend remotely immediately.

Execution of the *qsort* program is mostly stream AND-parallel. It looks as though it might also have independent AND-parallelism but this is bounded by the speed at which the *part/4* goals can split its input list into two output lists to be consumed by the two *qsort/3* consumer goals. To obtain speedup with a predominantly stream AND-parallel program, producers and consumers should execute on different processors. A consumer goal will reduce until it eventually exhausts its input stream and it will then suspend, sending a binding request to the producer goal asking for the rest of the stream. If the producer goal has generated more of the stream then the values will be sent and the consumer can continue work until it has used up the stream again and it will suspend sending another binding request. From this scenario we might expect a program executing in stream AND-parallel to generate a large number of binding requests. This certainly seems to be the case with *qsort* where the best strategy is the one that has both the best load balancing characteristic and the most binding requests, AP-NW.

Although the *primes* program is also a stream AND-parallel program, its behaviour is different from *qsort*: a pipeline of filter processes is created with integer values passed in on a stream at one end of the pipe and prime numbers passing out on a stream at

---

this is at most 4/25 = 16%.

the other end. Filter goals towards the end of the pipe are likely to be suspended most of the time and most of the processing activity will be at the end closest to the input stream of integers; the filter for multiples of 2, for example, will be active all of the time, but the filter for multiples of 131 will be mostly inactive. Most of the speedup through parallelism is going to come from the first few filter processes.

For *primes*, there is little difference in speedup characteristic between the distribution strategies, although NN-NW is the worst performer. Only the AP strategies generate a large number of binding requests, and although NN-W does not generate as many binding requests it does have the same speedup properties as the AP strategies. This, to an extent, contradicts the conclusions we came to about executing stream-AND parallel programs but may be because *primes* has most of its processes suspended for most of the execution time.

It is also not clear why there is no difference between a weighted and non-weighted AP strategy, but, for the NN strategies, adding weights does improve load balancing, suspensions, binding requests, and speedup.

The *queen-ls* program is another program that is stream AND-parallel based using the layered streams style of programming. We might expect the behaviour of *queen-ls* to be similar to the *qsort* program: good load balance and many binding requests indicate a good distribution strategy. Instead, load balance seems to be the best performance indicator in the that the AP strategies have better load balance and better performance than the NN strategies. With the AP strategies, however, AP-W has better performance than AP-NW but AP-NW has the better load balancing characteristics.

5.4 Summary

In this chapter we have presented the results of experiments with the FGHC interpreter. The results are from the execution of five test programs under different goal distribution strategies with varying numbers of processors. The results were analysed according to the measures we defined in section 3.7.

We have come to the following conclusions:

- The AP strategies give better performance than the NN strategies due to their better load balancing characteristics.
- The AP strategies are better at balancing load because spare work can diffuse more quickly across the machine than with the NN strategies.

- It is better to suspend a dependent goal locally than to allow it to be stolen by another processor on which it will suspend remotely.
- Ordering goals by weight had varying effect on program performance. Two programs showed no difference (*hanoi* and *primes*); two programs showed a slight performance improvement (*fib* and *queen-ls*); and one program showed a significant performance reduction (*qsort*).
- There was no consistent correlation between suspensions or binding requests and speedup.
- For the stream-AND parallel programs, *qsort, primes* and *queen-ls* there is strong correlation between suspensions and binding requests which suggests that most suspensions are on remote variables. This result is consistent with the producer-consumer nature of stream AND-parallel programs: a consumer will suspend, and send a binding request, when waiting for values from a producer.
- Although our measures are helpful at explaining some of the observed phenomena, there are still unexplained results. For example, for *primes*, why does the NN-W strategy perform as well as the AP strategies even though it has worse load balance?

Chapter 6

From interpreter to emulator

6.1 Introduction

In the previous chapter we presented the results for execution of an FGHC interpreter executing on a distributed memory parallel machine. The interpreter was executed evaluating five common benchmark programs under four different goal distribution strategies and for varying numbers of processors employed. The results included measurements of execution time, speedup, load balance, number of suspensions made, number of binding requests sent and number of goal ask messages sent. The results showed that there was a strong link between effectiveness at load balancing and speedup but little correlation between either suspensions or number of messages sent and speedup.

This lack of influence of suspension and message processing on the speedup characteristics of a program may have two causes:

1. The ratio of suspensions or messages processed compared to reductions made is very small.
2. The average time taken to make a reduction is much greater than that to make a suspension or to process a message.

We can test the hypothesis made in point 1 above by calculating the ratio of suspension or message counts to numbers of reductions. If we do this calculation for both suspensions and binding requests for the test programs we find that, for the programs *qsort*, *primes* and *queen-ls*, the total number of suspensions and the number of messages sent is of the same order of magnitude as the number of reductions made regardless of the goal distribution algorithm used.

Figure 6.1 shows plots of total reductions divided by average total suspensions for each

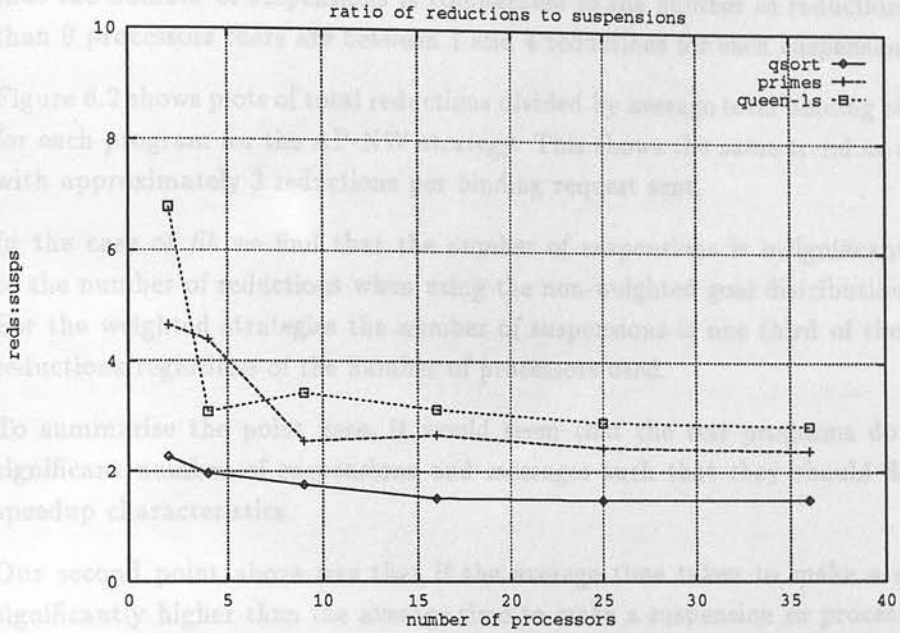

Figure 6.1: Ratio of reductions to suspensions

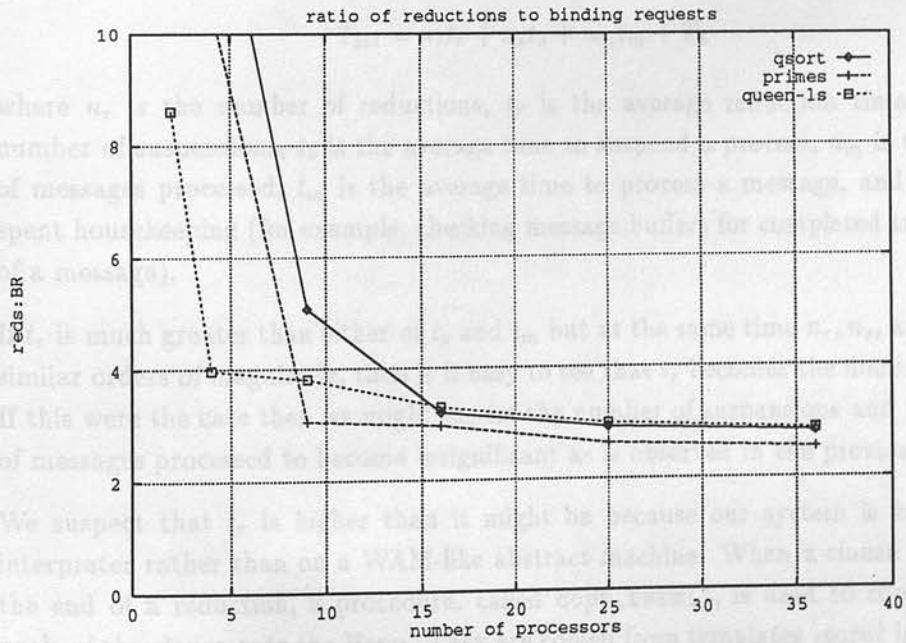

Figure 6.2: Ratio of reductions to binding requests

program for the AP-NW strategy (chosen to be a representative strategy). It shows that the number of suspensions is comparable to the number of reductions; for more than 9 processors there are between 1 and 4 reductions for each suspension.

Figure 6.2 shows plots of total reductions divided by average total binding requests sent for each program for the AP-NW strategy. This shows the same trend as suspensions with approximately 3 reductions per binding request sent.

In the case of *fib* we find that the number of suspensions is insignificant compared to the number of reductions when using the non-weighted goal distribution strategies. For the weighted strategies the number of suspensions is one third of the number of reductions regardless of the number of processors used.

To summarise the point here, it would seem that the test programs do generate a significant number of suspensions and messages such that they should figure in the speedup characteristics.

Our second point above was that if the average time taken to make a reduction is significantly higher than the average time to make a suspension or process a message then the time spent on reductions will swamp time spent elsewhere. Taking a single processor in a multiprocessor system, we can break down the total execution time of a program into constituent times spent in various parts of the system according to the formula:

$$T_{tot} = n_r t_r + n_s t_s + n_m t_m + T_h$$

where n_r is the number of reductions, t_r is the average reduction time, n_s is the number of suspensions, t_s is the average time to suspend a process, n_m is the number of messages processed, t_m is the average time to process a message, and T_h is time spent housekeeping (for example, checking message buffers for completed transmission of a message).

If t_r is much greater than either of t_s and t_m but at the same time n_r, n_s, and n_m have similar orders of magnitude, then it is easy to see that t_r becomes the dominant term[1]. If this were the case then we might expect the number of suspensions and the number of messages processed to become insignificant as is observed in the previous chapter.

We suspect that t_r is higher than it might be because our system is based on an interpreter rather than on a WAM-like abstract machine. When a clause commits at the end of a reduction, a procedure, called `copy_term()`, is used to copy the body goals of the clause onto the Heap. They are copied from templates stored in the clause store which is also part of the Heap. The procedure `copy_term()` is a universal term

[1]We are assuming that the time spent in housekeeping, T_h, is insignificant.

copier; it takes a term and constructs a copy on the Heap. This is inefficient since it is known before runtime how a new goal term can be specifically constructed. A WAM-like system does this by a sequence of **put** and **unify** instructions tailored to construct a particular term on the Heap. This specific instruction sequence is determined at compile time. Because of this specialisation, a considerable improvement in the average reduction time can be had by moving from an interpreter to a WAM-like system such as described in [Cra88]. If this improvement in average reduction time is realised, then the time spent on suspensions and message processing would be more prominent and may change the effectiveness of the goal distribution strategies.

Before moving from an interpreter to a WAM-like system it is as well to ask if it is possible to make corresponding improvements in the suspension and message processing areas of the system. If this is the case then some of the gains made by moving to a WAM-like system may be lost.

The time spent in processing messages can be divided up in the following way:

- packing the message into a linear array of characters comprising the message buffer;
- queueing the message buffer for transmission;
- sending the message;
- receiving the message into a message buffer;
- dequeuing the message buffer from the receive queue;
- unpacking the message.

Since we are using the T800 transputer processor which exchanges messages asynchronously on the communications links concurrently with the CPU computing, we can ignore the time spent in sending and receiving messages between processors. These are in any case fixed system parameters that the user cannot change. Similarly, queueing and dequeuing message buffers for transmission/reception is dependent on system calls to the MEIKO CSTools primitives and so cannot be reduced in time[2].

This leaves the time spent in packing and unpacking of message buffers. Message packing becomes significant when packing an arbitrary large term into a linearly arranged message buffer. The packing algorithm has to be general since the exact form of the terms to be sent between processors is not known until they have been fully packed.

[2]Performance could be improved by replacing CSTools, a general communications toolset, with a purpose built communications harness. To do so would be non-trivial so we assume that CSTools is an integral part of the computer that cannot be changed.

The implementation of the packing algorithm consists of a procedure that recursively traverses the term to be packed, copying portions of the term into the space left in the message buffer. This has been implemented fairly efficiently and it is doubtful that it could be significantly improved upon.

Time is also spent in unpacking terms. However, the packing algorithm was designed to make term unpacking as simple as possible; each cell in the message buffer is copied onto the Heap in turn, all the while checking the cells for internal relative pointers within the message which are translated to absolute pointers on the Heap.

All in all it would seem that the message processing routines are fairly well implemented and could not be easily improved upon.

Examining the mechanisms for suspending and awakening processes we come to a similar conclusion. During the attempted reduction of a goal, if it suspends on a variable in some candidate clause, then that variable is pushed onto the Suspension Stack. If the result of attempted reduction of the process is suspension then a hanger cell is made that points to the process record of the suspending process. Then each variable in the suspension stack is popped off in turn and a pointer to the hanger is added to the suspension queue attached to the variable. The suspension queues are implemented using normal linked list data structures and manipulation algorithms.

When a variable is assigned a value then its suspension queue is traversed and all the process records indirectly pointed to via hangers are rescheduled on the run queue. There seems to be no obvious way of decreasing the time spent on suspending goals or on reawakening goals.

The conclusion is that replacing the interpreter reduction mechanism with a WAM-like reduction mechanism is likely to significantly reduce the average reduction time but that the average suspension and message processing times cannot be easily reduced.

The rest of this chapter describes how the FGHC interpreter mechanism was replaced with a WAM-like reduction engine. We summarise the main reasons for making this change:

- To see if the suspension and message processing times are more important to the performance of a WAM-like implementation than they were in the interpreter based machine.
- To see how moving from an interpreter to a WAM-like machine affects the system performance. For example, will speedup characteristics improve or degenerate?
- To see if greater differences emerge between the various schedulers than were shown in the interpreter.

6.2 WAM-like emulators

With an interpreter, the source program is translated into Heap cells, and loaded into a special area of the Heap called the Clause Store. Goals are also represented as Heap cells on the Heap and new goals are made using a universal term copying procedure to copy body goals in the Clause Store onto the Heap.

With an emulator, the source program is compiled into a set of abstract machine instructions specific to evaluating the source program. FGHC systems designed around emulators usually use a modified WAM instruction set. There are two common ways of executing the WAM code produced by compilation:

- It is encoded as an array of characters (byte encoded) which are then loaded into an instruction interpreter. This interpreter has a library of routines, one for each instruction, which are called as the interpreter decodes each instruction in turn. This has the advantages of producing small code sizes for each program and is faster than the type of interpreter described previously.
- The code is further compiled to another target language, such as C, which is then compiled to the object code of the target machine. This produces stand alone object code for a given FGHC source program. This may give a speedup improvement over the former method because the instructions do not have an overhead due to decoding and because at each stage of compilation the compiler can make optimisations. The penalty is that a small source program can quickly expand to a large object code size.

For the FGHC emulator described here, the FGHC source is compiled to WAM-like instructions that are then converted to C instructions which are then compiled down to native code using a standard C compiler. This method was used because it was easier to implement than a byte code emulator and the large object code size was not a problem for the benchmark programs used in the experiments. It is as well to keep in mind that in a system that was to be used for substantial FGHC programs the code size could be a problem and, in that case, a byte code emulator might be a better choice.

The emulator system is described next. Many of the details of the emulator implementation are common to the interpreter described in chapter 4. The emulator is described in terms of the modifications made to the interpreter to implement it, and the form of the abstract machine instructions and machine architecture.

6.3 The Emulator

The Warren Abstract Machine was invented by Warren[War83] as a machine representation which is easy to compile Prolog programs down to while at the same time easy to implement on a conventional register and stack based computer.

The WAM and its variants will not be described here. For a description of the WAM for Prolog see[Ait90]; for a description of a WAM-like abstract machine implementation for committed choice languages see Crammond's work[Cra88].

The machine we describe here is partly based on ideas gained from all the above sources, especially Crammond [Cra88], but is also influenced by work by Foster and Taylor on the design of Strand88[FT90], Kimura and Chikayama[KC87] on the design of a KL1 system[3], and researchers working on the Janus project[SKL90].

6.3.1 Data structures

6.3.1.1 Heap Cells

The interpreter has two special cell types which are not needed in the emulator: **SYS** and **REG**.

SYS was used to distinguish between Heap cells which will be interpreted as system goals and Heap cells which are plain functor cells. This cell type is no longer needed in the emulator. System goals are represented explicitly in the program's abstract code rather than implicitly in Heap cells.

REG was used to mark cells which refer to arguments (or variables) in the cell representation of a clause template on the Heap. This cell type was used when copying the clause template for generating new sub-goals; the arguments for the sub-goals were copied from the arguments in the environment of the current goal via the **REG** cells. In the emulator, constructing sub-goals is made explicit through special instructions — clause templates are no longer used — and so the **REG** cell type is redundant.

Three new cell types were introduced in the emulator: **FRE**, **LIS**, and **NIL**.

FRE marks a free cell and is used to reclaim space in certain circumstances (explained below in the Argument Stack section).

LIS introduces a special cell type to mark the start of a list cell. This cell is followed by two cells, the **car** and **cdr** parts of the list cell. In the interpreter, this was implemented

[3]KL1 is FGHC augmented with a control language for controlling goal execution so that it can be used to implement computer operating systems.

implicitly using a cell <STR, ./2>.

NIL introduces a special cell type to mark the empty list. In the interpreter, this was represented implicitly by the cell <CON, []>.

The new complete table of cell types and forms is as follows:

tag	*data*
VAR	pid, heap index
SUSP	sl_index
STR	functor_id, arity
CON	functor_id, 0
INT	integer
LIS	
NIL	
FRE	

6.3.1.2 Registers

For our WAM-like machine we need to introduce a number of additional registers to those defined in the interpreter.

Registers in the interpreter are named explicitly as operands to instructions. In the WAM, argument registers are referred to as A registers and registers used to hold temporary values are referred to as X registers. These registers are overlayed — A1 refers to the same location as X1 — and so they are referred to here simply by an integer starting from 0. The WAM also defines Y registers which are allocated in the environment of the goal. FGHC does not need environments and so there are no Y registers.

The S register, which holds a Heap address, is used when input matching a sub-argument of a structure. This register is implicitly referenced by the **read** instructions described below.

The BP register is used to hold the address of the next clause to be attempted in the relation if the current clause does not succeed.

The TS register holds the maximum number of reductions to be made before the next **time slice**. This register is decremented after each reduction. If it becomes zero then program control returns from the user code to the Toplevel so that messages from other processors can be processed; it is equivalent to the **reductions_per_cycle** mechanism used in the interpreter. This is not time slicing in the usual sense where it is used to give another process a chance at reducing to ensure **fairness**. The emulator (and indeed the interpreter) does not guarantee fairness and the time slicing is a mechanism

for dividing system time between reducing goals and performing other system tasks. TS is reset to a fixed value after reaching zero.

The P register, used in the WAM to point to the next instruction to be performed, is not used in this emulator. This is because the WAM-like instructions are ultimately compiled to native code and the program counter of the target processor is the equivalent of the P register.

6.3.1.3 Argument Stack

In the interpreter the arguments of a goal were represented as cells on the Heap. This is wasteful of Heap cells because, although argument cells are easy to mark for garbage collection, implementing a general garbage collection algorithm is difficult in a parallel environment, and so if a garbage collector is not implemented, as is the case here, then argument cells that are no longer referenced take up valuable Heap space.

An improvement over the interpreter is to partition the cell space into a space for true Heap cells, or data, and a space for argument cells. This improvement was resisted in the interpreter because packing and unpacking terms as messages are more complicated; with all cells existing in one space it is possible to use the same routines (packing/unpacking) to send a cell binding or to send a goal description but with the cells partitioned into argument cells and data cells slightly different routines are needed for the packing/unpacking to distinguish between sending/receiving a goal or data.

On moving to the WAM-like machine it was decided that this extra level of complexity of sending messages was justified: the cell space was partitioned into a space for the arguments of goals — the Argument Stack — and the Heap. The Argument Stack is modelled on that implemented by Crammond[Cra88]. There he describes a space of cells that behaves like a stack: cell values are allocated from the top of the stack. The Argument Stack does not behave like a stack in that cells may also be deallocated from inside the stack so creating holes, that is areas which are not allocated but are below the top of the stack. This is necessary because in a committed-choice language machine, unlike a Prolog machine, goals are not guaranteed to terminate in a specific order and so it cannot be determined in what order the arguments of terminated goals will be deallocated. Crammond noted, however, that many programs do use the Argument Stack in a stack-like manner and that often holes do not persist for very long.

More fully, the Argument Stack is an array of cells used to hold the arguments of non-terminated goals. A pointer to the top element of the Argument Stack is held in the special register TAS. When a process is created the arguments of the goal are allocated on the Argument Stack. When a goal has terminated, the argument cells are deallocated from the Argument Stack. If the cells to be deallocated are at the top of

the Argument Stack then the TAS register is adjusted downwards. If the cells are not at the top of the stack then they are converted into cells of type FRE, that is empty cells. If at any time the TAS register is found to be pointing to a cell of type FRE then it is decremented until it is pointing to a cell not of type FRE; it is then incremented by one cell so that it is pointing to the next free cell for allocation.

To accommodate this change, the message packing/unpacking routines were modified so that when sending goals packing could commence from the argument stack instead of the Heap, and so that unpacking would result in the arguments of the transmitted goal being reformed on the Argument Stack of the receiving machine.

6.3.1.4 Process records

In the interpreter, a process record pointed to the representation of a goal, stored as cells on the Heap, through the **goal ptr** field. As we have seen, goals are no longer stored on the Heap and their arguments are stored on the Argument Stack. This introduces two new fields into the process record structure: **argp**, a pointer into the Argument Stack of the start of the arguments for this process; and **nargs**, the number of arguments used by this process.

Another field that is also now needed is a pointer to the address of the entry point into the code to be executed when running the goal, called **codep**, which replaces the **goal ptr** field in the interpreter. At first glance, **codep** will need to store a reference to an address in the stand alone program code. In C, places in a C program are marked by labels, but in standard C it is not possible to store labels in a variable, which makes this field seem impossible to implement. One can get around this problem, however, by a non obvious programming trick, described in [GDBD92], which will be described later.

6.3.1.5 Miscellaneous

It is worth describing the parts of the interpreter that are removed when moving to the emulator. The major omission is that the program clauses are no longer encoded as cells and stored on the Heap. In the emulator the program is embodied in WAM-like instructions and ultimately in stand alone machine code; the Heap is reserved for data generated during computation.

Another consequence of removing the program code from the Heap is that the Symbol Table no longer maintains pointers into the start of a list of relevant clauses attached to symbols; the **firstclause** and **lastclause** fields of a symbol table entry are omitted. Also the **weight** field is not needed as goal weights are defined in instruction operands.

In fact, most of the symbol table can be omitted and the remaining table is used only for looking up the print string of functors when printing terms.

6.3.2 Abstract machine instructions

In the interpreter, the FGHC program clauses are encoded as cells on the Heap. In the emulator, the FGHC program is compiled to a list of WAM-like instructions. The instructions are defined as C macros and the program is ultimately expanded by the C preprocessor before being compiled to native code.

Before describing the abstract instructions used by the emulator it is as well to show the form of the abstract code generated when an FGHC clause is compiled. The general form of an FGHC clause is:

$$H(a_1, \ldots, a_m) : -G_1, \ldots, G_n : V_1 = T_1, \ldots, V_p = T_p, B_1, B_2, \ldots, B_q.$$

where the clause belongs to the relation H/m, a_1 to a_m are the formal parameters of the head, G_1 to G_n are guard goals, $V_1 = T_1$ to $V_p = T_p$ are output unifications, and B_1 to B_q are body goals.

The general case clause will be compiled into an abstract program of the form:

```
input matching on a1
...
input matching on am
code for guard G1
...
code for guard Gn
output unification V1 = T1
...
output unification Vp = Tp
construct arguments for goal Bq
schedule goal Bq
...
construct arguments for goal B2
schedule goal B2
construct arguments for goal B1
execute goal B1
```

The execution of a clause starts with an input matching phase where the input arguments of the goal are input matched against the clause's arguments. There then

follows a guard execution phase which may consist of some input matching, some tests on arguments, or some arithmetic calculations. If the above phases succeed then execution continues to the output unification phase where output variables are assigned terms. This is followed by instructions that construct the arguments for each body goal, create a process record, and schedule the body goal on the run queue. Notice that the goals are constructed and scheduled in reverse order in this example. This is because the Goal Queue may be used as a stack and scheduling goals (pushing them onto the 'stack') in reverse order means that the order of execution of goals is the same as the left-to-right ordering of goals in the original FGHC clause. The final goal to be constructed will then be the next one to be executed and so its arguments are constructed and process record are constructed, and the process is immediately executed without needing to be scheduled on the run queue.

From this general template of a compiled clause we can see that we need the following types of instructions:

- input matching;
- arithmetic;
- tests;
- output unification;
- term constructors;
- scheduling and control instructions.

The majority of the abstract machine instructions are listed in the table in figure 6.3. As a key to the instruction operands: *r, r1 and r2* are registers; *c* is a constant; *i* is an integer constant; *f/a* is a functor '*f* with arity '*a*'; *L* is a label; *w* is an integer weight.

We will not describe the instructions in detail here since they have been covered in similar machines by other researchers. These instructions are a subset of Crammond's JAM[Cra88] but instead of using the usual WAM **unify** instructions, the **set** instruction notation of Ait-Kaci[Ait90] has been preferred.

6.4 Implementation

The emulator is based on the interpreter. Instead of storing the FGHC program as cells on the Heap, it is compiled into abstract instructions. The abstract instructions are implemented as C macros that operate on the abstract data structures, for instance,

Input Matching			
wait_list(r) wait_struct(f/a,r)	wait_const(c,r) wait_val(r1,r2)	wait_int(i,r) wait_var(r1)	wait_nil(r)
read_const(c) read_int(i)	read_nil read_void	read_var(r)	read_val(r)
Term Constructors			
put_list(r) put_str(f/a,r)	put_const(c,r) put_val(r1,r2)	put_int(i,r)	put_nil(r)
push_list push_str(f/a)	push_const(c) push_val(r)	push_int(i) push_var(r)	push_nil push_void
set_nil set_var(r)	set_const(c) set_val(r)	set_int(i)	set_void
Output Unification			
get_nil(r) get_list_val_var(r1,r2,r3) get_list_var_var(r1,r2,r3)	get_const(c,r)	get_int(i,r)	get_val(r1,r2)
Tests			
eq_val(r1,r2) neq_val(r1,r2) integer(r)	eq_int(i,r) neq_int(i,r)	less_val(r1,r2) lesseq_val(r1,r2)	less_int(i,r) lesseq_int(i,r)
Arithmetic			
add_val_val(r1,r2,r3) sub_val_val(r1,r2,r3) mul_val_val(r1,r2,r3) div_val_val(r1,r2,r3) mod_val_val(r1,r2,r3)	add_int_val(i,r1,r2) sub_int_val(i,r1,r2) mul_int_val(i,r1,r2) div_int_val(i,r1,r2) mod_int_val(i,r1,r2)	 sub_val_int(r1,i,r2) div_val_int(r1,i,r2) mod_val_int(r1,i,r2)	
Control and Scheduling			
switch_on_term schedule(f/a, a, w) label(L)	trust_me_else(L) execute(f/a, a, w)	suspend otherwise	proceed

Figure 6.3: Abstract machine instructions

the Registers, the Heap, and the Argument Stack. Let us call the abstract instructions that implement the user program the **abstract program.**

Taking the interpreter as the reference system, all the routines for storing the program as cells, interpreting the program, and copying goal templates to make new goals, have been removed. What is left is the Toplevel code, the routines that implement the Communications Manager and Scheduler, and a library of useful routines for such things as dereferencing, term unification, and suspending/awakening goals. This code is common to each FGHC user program and can be precompiled to form an object code library.

```
#include "macros.c"

enum {main_0_1=0, ..., <Labels>};
enum {<Constants> ...};

run()
{
 top:
  switch(nxt_lbl){
    label(main_0_1);
      .      .
      .      .
      [abstract program goes here]
      .      .
      .      .
  }
}
```

Figure 6.4: C wrapper for abstract program

6.4.1 The switch trick!

The abstract program is put inside a C wrapper, to define a special routine called `run()`, shown in figure 6.4.

The Toplevel switches between executing the user code (now implemented as `run()`) and the Communications Manager, just as it did in the interpreted system. This program with wrapper is compiled using a C compiler and linked together with the object code library. The result is that, for each FGHC user program, standalone object code is produced that will execute on a sequential or distributed memory parallel computer.

Why is the wrapper needed? The FGHC program is compiled down ultimately to standalone object code which is executed by the target machine just like any other

```
#define label(lab)\
  case lab:\
  lab ## _l:

#define trust_me_else(lab)\
  BP = lab;

#define wait_list(reg)\
{\
  Addr Ta;\
  Ta = deref(REGISTERS+reg);\
  switch(type_of(Ta)){\
  case LIST:\
    S = Ta+1;\
    break;\
  case VARIABLE:\
  case SUSP:\
    push_sstack(Ta);\
    update_state(SUSPEND);\
    nxt_lbl = BP; goto top;\
  default:\
    update_state(FAILURE);\
    nxt_lbl = BP; goto top;\
  }\
}
```

Figure 6.5: Example C macros for abstract instructions

executable object code. But this approach creates one large problem: how to set up goals that point into the various entry points for clauses in the program. For example, when executing the *hanoi* program, how do we set up a process record for a *move/4* process? The process record will somehow need to point to the entry point in the object code for *move/4*.

One approach would be to put a C label at each entry point in the C code, and then store the label as an entry in the process record. The problem with that is that the ANSI C definition does not allow the use of C labels as first class objects[4]: there is no way of calculating the address of the object code referenced by a label.

A clever way of getting around this problem is mentioned in a paper on the efficient implement of Janus [GDBD92]. The idea is to enclose the whole code in a `switch` statement which switches on a global integer variable `nxt_lbl`, and the start of the `switch` is referenced by the special label `top`. Entry points in the code are referenced by both a `case` statement and a C label. This gives two options for jumping to a

[4]Gnu GCC version 2 does allow C labels to be first class objects through an extension to the GCC compiler. The extension, however, is not part of ANSI C and is not portable across compilers.

certain point in the code:

```
goto < label >;
```

jump straight to the entry point in the program, or:

```
nxt_lbl = < case_number >; goto top;
```

goto the entry point indirectly by jumping to the beginning of the `switch` and switching on `nxt_lbl`.

This approach has the advantage that the case statement parts of the switch are integers so an entry point can be stored in a process record as an integer value.

6.4.2 Example macros and program

To illustrate the discussion of the emulator implementation we show some of the C macros used to implement various abstract instructions, and show a typical result of compiling an FGHC program into C.

Figure 6.5 shows the C macros used to implement `label(L)`, `trust_me_else(L)`, and `wait_list(R)`. The instruction `label(L)` is simply translated into `case L:` and into a label of `L` with the suffix `_1`. (The ANSI C preprocessor directive `A ## B` constructs a new string from the concatenation of the string in `A` with the string in `B`.) This enables the labelled point to be jumped to either indirectly via the `switch` and `case`, or directly via the usual C label.

The instruction `trust_me_else(L)` shows that the 'label' can be stored for use after a subsequent instruction failure.

Figure 6.6 shows the result of compiling the *hanoi(15)* FGHC program into an abstract program with C wrapper around it. FGHC programs are currently 'compiled' into this form by hand. The code generated is what would expected from a reasonable, but not too clever, compiler and is comparable with the code used by other researchers (for example, Crammond [Cra88]).

6.5 Performance

Figure 6.7 shows the execution times of the test programs, when executed on a single processor, for both the interpreter and the emulator. There has been a significant reduction of execution time in moving from the interpreter to the emulator; a speedup factor of between 5 and 8 is observed.

```
#include "macros.c"

enum {main_0_1=0, hanoi_1_1, move_4_1, move_4_2, move_4_3};
enum {c_left, c_center, c_right};

run()
{
 top:
  switch(nxt_lbl){
    label(main_0_1);
      push_int(15);
      call(hanoi_1_1,1,2);
      proceed;

    label(hanoi_1_1);
      put_const(c_left,1);
      put_const(c_center,2);
      put_const(c_right,3);
      execute(move_4_1,4,2);

    label(move_4_1);
      trust_me_else(move_4_2);
      integer(0);
      neq_int(0,0);
      sub_int(1,0,4);
      push_val(4);
      push_val(3);
      push_val(2);
      push_val(1);
      call(move_4_1,4,2);
      put_val(4,0);
      put_val(2,5);
      put_val(3,2);
      put_val(5,3);
      execute(move_4_1,4,2);
    label(move_4_2);
      trust_me_else(move_4_3);
      integer(0);
      eq_int(0,0);
      proceed;
    label(move_4_3);
      suspend;
  }
 }
```

Figure 6.6: C code for compiled *hanoi(20)* program

		Interpreter		*Emulator*		
Program	*Reductions*	time(s)	LIPS	time(s)	LIPS	E/I
hanoi(15)	65537	48.5	1350	8.0	8192	6.2
fib(20)	32837	35.0	938	7.0	4691	5.0
qsort(1024)	11543	20.0	577	2.9	3980	7.0
primes(800)	22730	29.3	776	3.7	6143	8.1
queen-ls(8)	23627	45.3	521	6.9	3423	6.7

Figure 6.7: Timing comparison between the interpreter and emulator for a single T800 processor

	Strand88		*Emulator*		
Program	time(s)	kLIPS	time(s)	kLIPS	E/S
hanoi(15)	0.93	70	0.88	74	1.1
fib(20)	1.73	23	0.63	52	2.3
qsort(1024)	1.87	34	0.25	46	1.4
primes(800)	0.76	31	0.30	76	2.5
queen-ls(8)	1.48	24	0.53	44	1.8

Figure 6.8: Timing comparison between Strand88 and the emulator for a single Sun4 processor

Figure 6.8 is a comparison of the execution times of the emulator (compiled under GCC-2.3.3 with optimisation `-O2`) and Strand88 (Buckingham Release Version B6) both executing on a Sun4 SPARC processor. It shows that the emulator is quicker than Strand88 by between 1.1 and 2.5 times, although comparison of systems is difficult.

A final comparison is with the Janus emulator `jc` described in [GDBD92] which executes a `hanoi(13)` program in 283 ms (unoptimised) and 182 ms (optimised) on a Sun4 SPARCstation 1 using the Sun C compiler `cc`. The execution time for the emulator (compiled under GCC-2.3.3) is 216 ms which is part way between the unoptimised and optimised results for `jc`, although the *hanoi* programs may not be the same; the Janus benchmark program's source is not published with the paper.

6.6 Summary

This chapter began by hypothesising why the numbers of suspensions and messages generated during execution of the test programs with the FGHC interpreter did not correlate with execution time.

The first hypothesis was that not enough suspensions and messages are generated to

affect program execution. An analysis was made of suspension counts and binding request messages sent for each of the test programs, and it was shown that, compared to numbers of reductions, there are significant numbers of suspensions and messages generated.

The second hypothesis was that the average time for a reduction in the interpreter is so large that time spent in suspending goals and sending/receiving messages is insignificant by comparison. Reduction times are large because a universal term copying routine is used to create new goals after a successful reduction. A WAM-like instruction emulator provides substantial improvements in reduction times by executing sequences of term constructor instructions that are tailored to creating a specific goal.

The rest of the chapter presented the design and implementation of a WAM-like emulator system with which to reduce the average reduction time. The emulator is based on the interpreter but with the general interpreting reduction engine replaced by an abstract program of abstract instructions. The WAM-like abstract instructions are implemented as C macros which are macro expanded by a C precompiler. The result of this is compiled together with a library of supporting procedures that provide the functionality for such things as communication between processors, and suspending and resuming goals. The WAM-like instruction set used in the emulator is a subset of those used by Crammond for his shared memory implementation of the committed choice languages.

The single processor execution of the emulator was evaluated using the test programs. It executes at 6–8 times faster than the interpreter.

Finally, a comparison in execution time was made between the emulator, Strand88 and `jc` (an implementation of Janus). Executing on a single Sun4 SPARC processor, the emulator compared well with Strand88, executing between 1 and 2.5 time faster than Strand88 for the same test program. The execution time for the emulator executing the *hanoi* program is of the same order as the execution time of a similar program reported for `jc`.

Chapter 7

Results — Emulator

7.1 Introduction

This chapter presents the results of executing the test programs, under the various goal distribution strategies, using the emulator described in the previous chapter.

The relationship of the results of the emulator and the interpreter are discussed and general conclusions drawn.

7.2 Experimental procedure

The experimental procedure was the same for the experiments with the interpreter: the number of processors, test program, and goal distribution strategy were varied with ten runs made of each configuration; statistics were gathered for each run and average measures produced.

The only component that has changed from the interpreter system when using the emulator system is that the average time to make a reduction has reduced. All other parameters, for example the number of reductions done before checking for messages, the topology of the target machine, and the size and nature of the test computations, were kept the same.

The idea is to compare the two sets of results in the light of the change in the average time for a reduction.

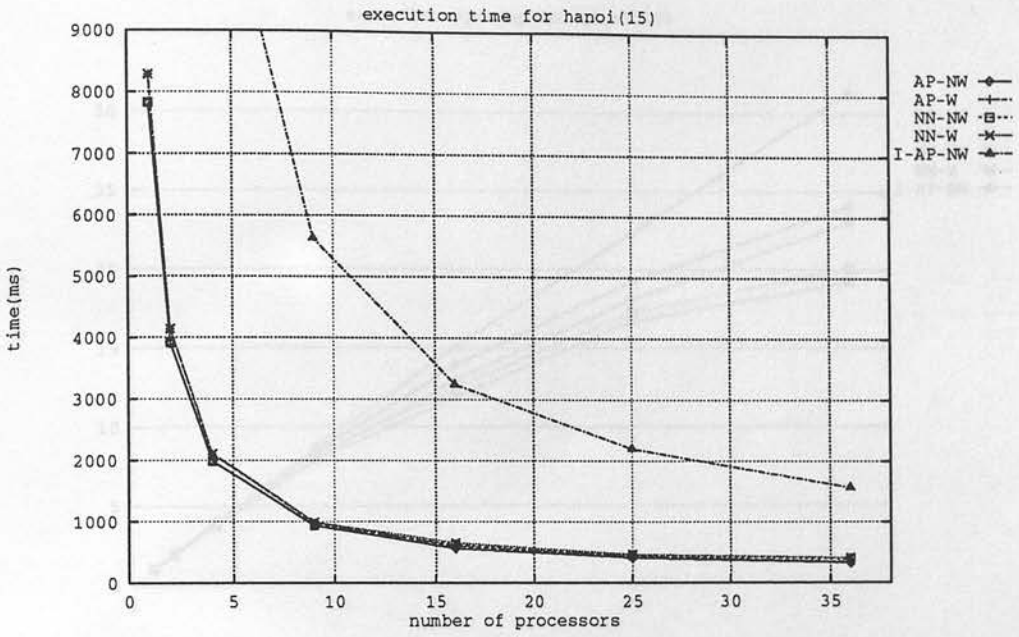

Figure 7.1: Emulator: Execution time for *hanoi(15)*

7.3 Results

7.3.1 hanoi

Figure 7.1 shows execution time characteristic for *hanoi* using the emulator system. A plot of the AP-NW strategy from the interpreter results is also shown (labelled I-AP-NW) for comparison. It is clear that switching to the emulator has resulted in a significant reduction in execution time over the interpreter: at 1 processor, the ratio of I-AP-NW to AP-NW is 6.3; at 36 processors the ratio is 4.8. The emulator is 5 to 6 times faster than the interpreter over the range of processors tested. This reduction in execution time is due to the improvement in the average reduction time of the emulator compared to the interpreter.

Figure 7.2 shows the speedup characteristic for *hanoi*. The goal distribution strategies form two groups: the AP strategies perform better than the NN strategies. The weighted version of a distribution strategy also performs worse than the non-weighted version. This is due to the extra time taken in checking the weight of a goal as it is put onto the goal queue.

A plot of the best strategy for the interpreter, I-AP-NW, is also shown on the graph. It shows that there has been a reduction in terms of speedup in moving from the interpreter to the emulator. At 36 processors, I-AP-NW has a speedup of 31, whereas

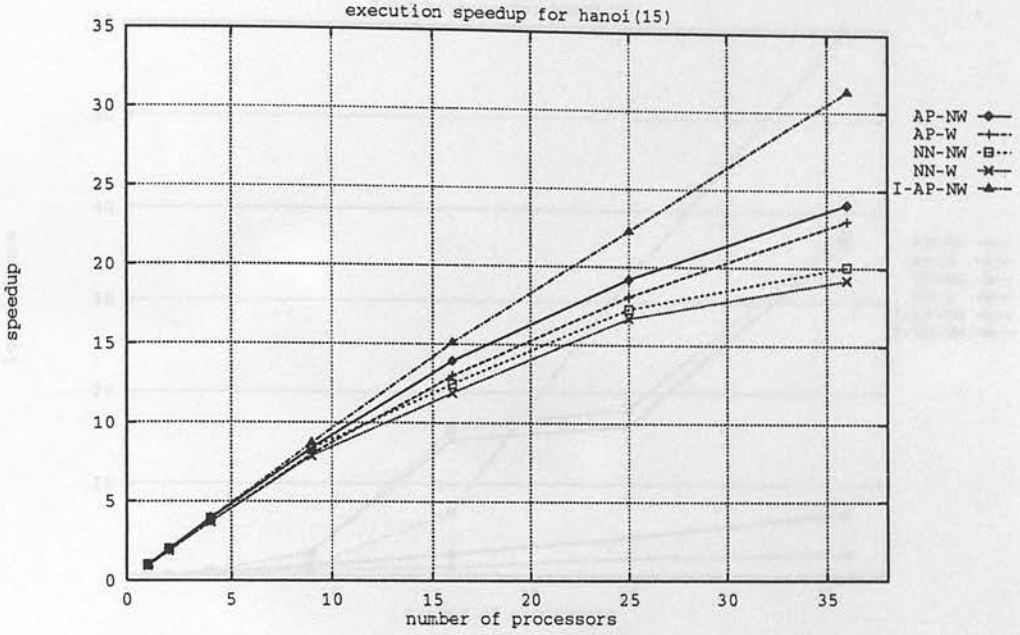

Figure 7.2: Emulator: Speedup for *hanoi(15)*

AP-NW has a speedup of 24. The difference between the AP and NN strategies has narrowed in the emulator results when compared with the interpreter results. The speedup characteristics for the emulator AP strategies have become worse when compared to the interpreter AP strategies, whereas the speedups for the emulator and interpreter NN strategies are about the same (not shown on the graph).

The most likely reason for the worsening speedup characteristics of the emulator AP strategies over the interpreter AP strategies is that the lessening in the average time for a reduction has increased the prominence of the time taken in scheduling goals and sending messages. In other words, the amount of work generated by sending out a *move/4* goal to a remote processor has been significantly reduced, there will be many small goals that generate very small amounts of work compared with the expense of sending them out for remote execution.

The same worsening effect was not observed in the NN strategies because their speedup curves seem to be limited by the rate at which goals can be distributed over the mesh, and that has not changed significantly from the interpreter to emulator versions.

Figure 7.3 shows the load balancing characteristics of *hanoi*. Again the distribution strategies form two groups: the AP strategies having significantly better load balancing characteristics than the NN strategies.

Also plotted are two of the results for the interpreter experiments: I-AP-NW and

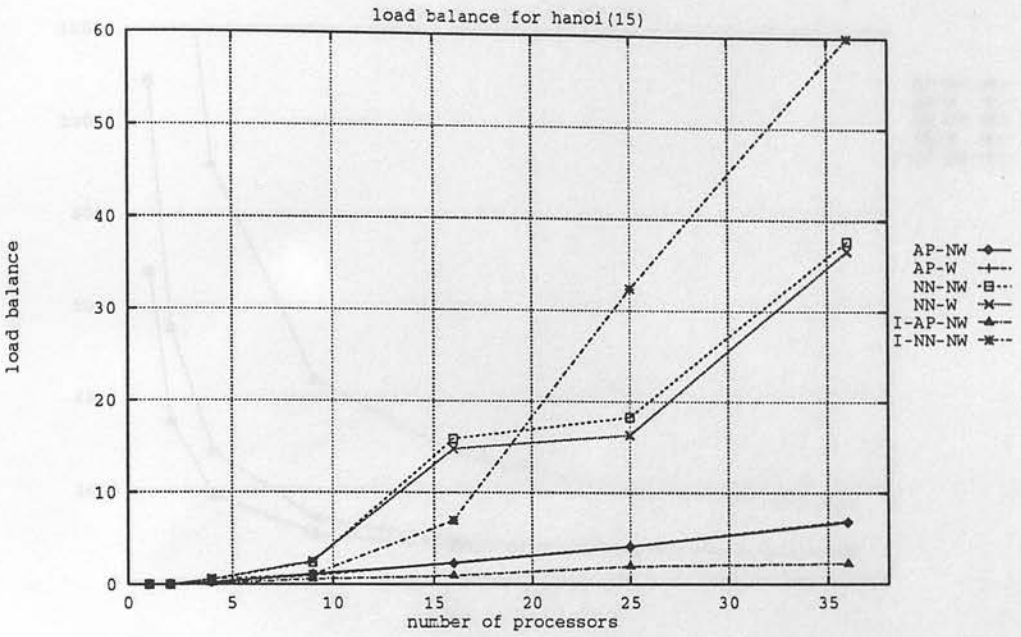

Figure 7.3: Emulator: Load balancing for *hanoi(15)*

I-NN-NW. Firstly, we note that these strategies have the same distinctions between them; I-AP-NW has significantly better load balancing characteristics than I-NN-NW. Secondly, the emulator AP strategies have worse load balancing characteristics than the respective interpreter strategies. But conversely, the emulator NN strategies have better load balancing characteristics than the respective interpreter strategies. For the emulator, the AP strategies have become less responsive and the NN strategies have become more responsive.

The reason for the AP strategies becoming less responsive is likely to be because of the reduction in grain size of goals. In fact, the responsiveness of the strategies has not reduced but the reduction time is smaller and to compensate the responsiveness would need to be increased accordingly. But it has stayed the same, and so balancing the reduced load is done less well.

The emulator NN strategies are better at load balancing than their interpreted counterparts. This probably shows that the rate at which goals diffuse over the mesh has increased slightly — the emulator NN strategies have become more responsive — but there is still a large difference between the load balancing characteristics of the emulator NN and AP strategies; the increase in responsiveness does not make up much ground in load balancing even though the emulated AP strategies have degenerated compared with the interpreted versions.

Suspensions and binding requests are identical for the interpreter and emulator results:

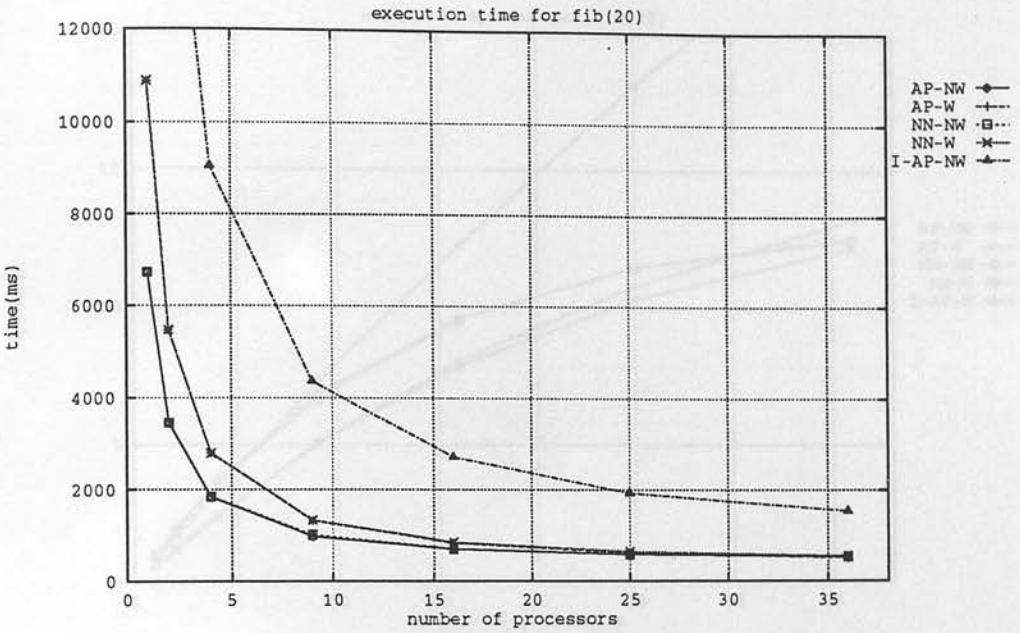

Figure 7.4: Emulator: Execution time for *fib(20)*

they are zero since the *hanoi* program generates no suspensions or binding request messages.

To summarise:

- The AP strategies have better speedup characteristics than the NN strategies.
- This is due to the better load balancing characteristics of the AP strategies over the NN strategies.
- The emulator AP strategies have worse load balancing characteristics than the respective interpreter strategies. This is likely to be due to the reduced grain size of goals distributed over the computer.
- The emulator NN strategies have better load balancing characteristics than the respective interpreter strategies. This is likely to be because the responsiveness of the former has increased slightly compared to the latter.
- The emulator strategies execute between 5 and 6 times faster than the interpreted strategies.

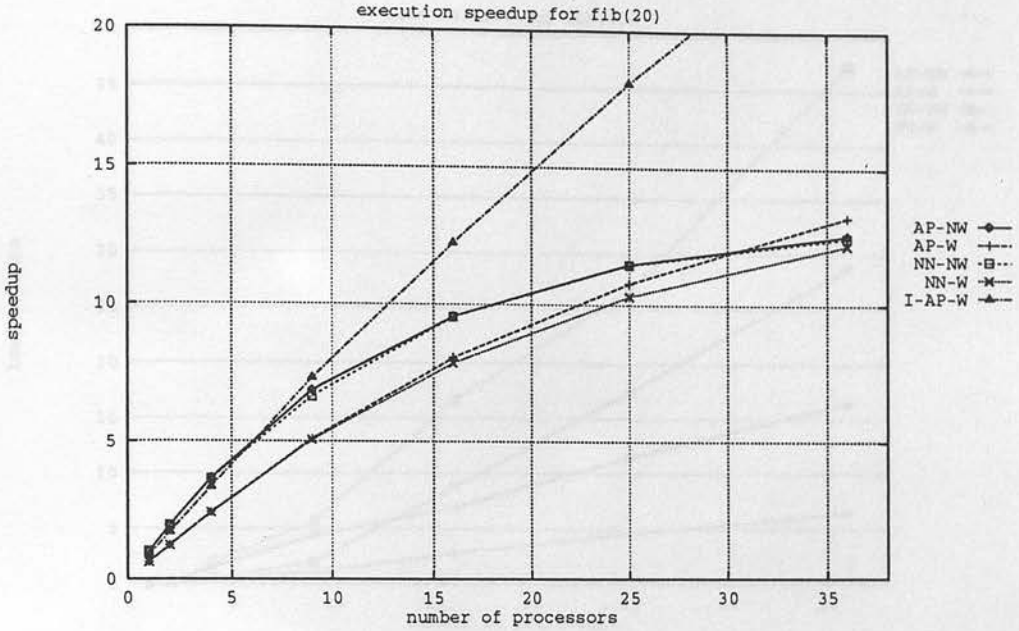

Figure 7.5: Emulator: Speedup for *fib(20)*

7.3.2 fib

Figure 7.4 shows the execution time characteristic for *fib*. For small numbers of processors, the weighted strategies (AP-W and NN-W) have a worse execution time than the non-weighted strategies (AP-NW and NN-NW). From knowledge gained from the interpreter experiments, this is because of the large numbers of local suspensions that the weighted strategies generate. As the number of processors increases, however, the differences between the strategies becomes smaller, until at 36 processors the different plots become indistinguishable.

The result of the interpreter strategy I-AP-NW is also plotted. The emulator strategies show a significant decrease in execution time compared to the I-AP-NW: at 1 processor a 5 fold decrease, and at 36 processors a 3 fold decrease. This also shows that the advantage the emulator has over the interpreter becomes less as the number of processors is increased.

Figure 7.5 shows the speedup graph for *fib*. For from 1 to 25 processors, the strategies fall into two groups: the non-weighted strategies performing better than the weighted strategies. This effect was seen in the results for the interpreter although it was much less significant; it only occurred between 1 and 16 processors and there was a much less marked difference between the groups. The difference between the weighted and non-weighted strategies is that the weighted strategies generate a large number of

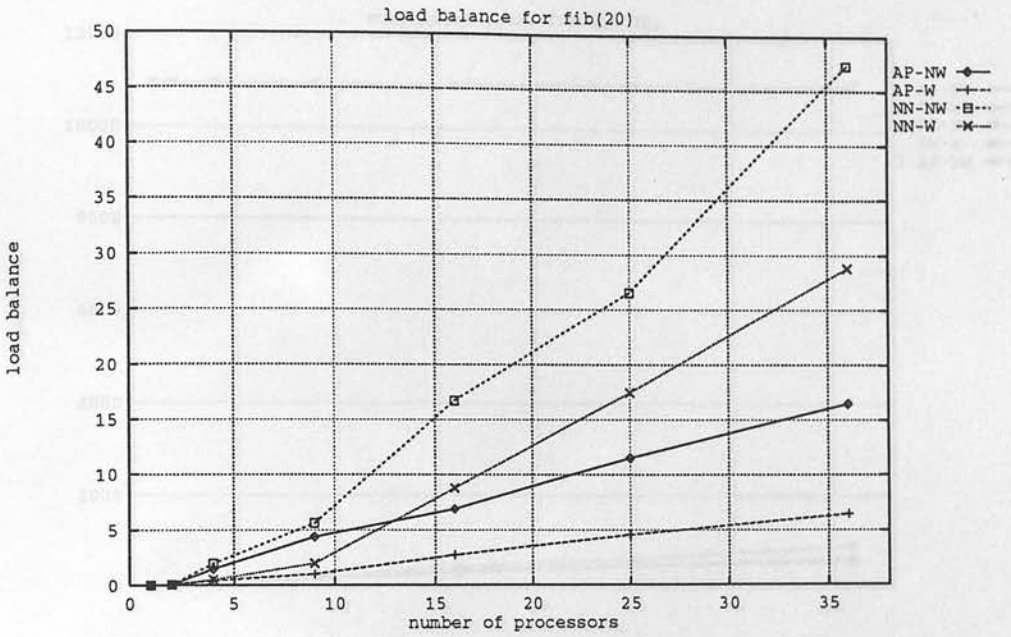

Figure 7.6: Emulator: Load balancing for *fib(20)*

suspensions from suspending *add/3* goals locally. The increase in the effect of the local suspensions is due to the decrease in the prominence of the average time to reduce a goal; the time taken to suspend a goal has become more significant in the emulator compared with the interpreter.

For more than 16 processors, the weighted strategies begin to have better speedup values than the non-weighted strategies, with AP-W having the best speedup value at 36 processors. A similar effect was noted with the interpreter results, although it started earlier at 16 to 25 processors. As we noted with the interpreter results, the effect is likely to be because the non-weighted strategies are distributing *add/3* goals which suspend immediately, wasting time in distributing useful work. In contrast, the weighted strategies suspend the *add/3* goals early which eventually pays dividends in better and more productive load balancing.

Also plotted on the graph is the speedup characteristic for the best strategy from the interpreter results, namely I-AP-W. The speedup results for the emulator are much worse than those for the interpreter. This again shows that as the grain size of the computation is reduced (in this case by reducing the average time of a reduction) then it is harder to obtain good speedup characteristics.

Figure 7.6 shows the load balancing characteristics for *fib*. From 1 to 9 processors, the strategies form into two groups: the weighted strategies do better at load balancing than the non-weighted strategies. From 16 to 36 processors, however, the strategies

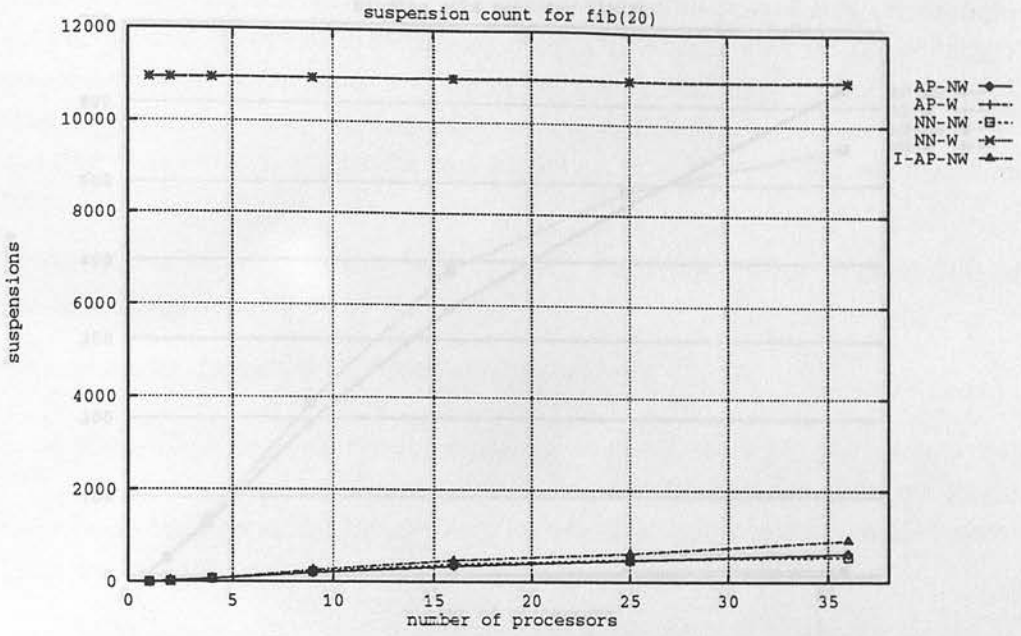

Figure 7.7: Emulator: Suspensions for *fib(20)*

group themselves into the more usual AP group and NN group, with the NN strategies performing worse than the AP strategies. Again, in common with the interpreter, the weighted strategies have better load balancing characteristics than their non-weighted equivalents.

When comparing the emulator and interpreter results, the NN strategies have about the same load balancing characteristics for both, but the AP strategies perform worse at load balancing in the emulator than in the interpreter. This is consistent with the findings of *hanoi* above: when comparing interpreter and emulator results, the NN strategies have the same or slightly improved load balancing, but the AP strategies have significantly worse load balancing in the emulator.

One area where not much difference is expected between the results from the interpreter and emulator is in the number of suspensions generated. The interpreter results show that the weighted strategies cause *add/3* goals to suspend early locally generating a large number of local suspensions (11,000); the non-weighted strategies generate many fewer suspensions (1000) which are due to *add/3* goals being distributed and suspending remotely.

Figure 7.7 shows the suspensions generated by the emulator executions of *fib*. This confirms expectations in that the weighted strategies generate 11,000 suspensions and the non-weighted strategies generate many fewer suspensions. The results for I-AP-NW from the interpreter are also plotted; they show that the non-weighted emulator

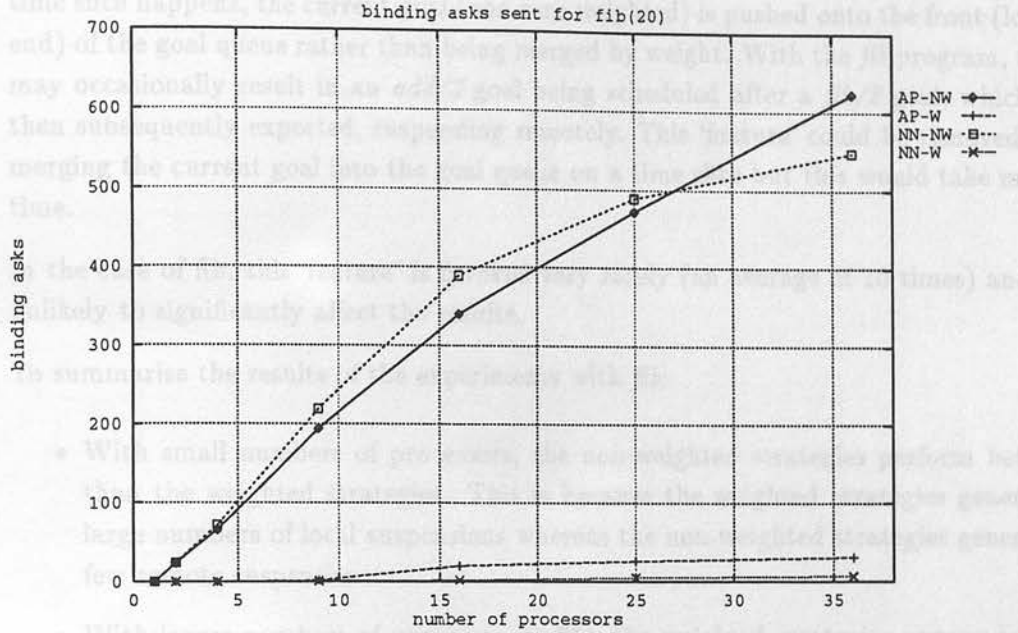

Figure 7.8: Emulator: Binding requests for *fib(20)*

strategies generate fewer suspensions than the interpreter equivalent. But apart from this minor difference the results from the interpreter and emulator are very similar.

It is also expected that the binding request characteristics for the interpreter and emulator will be similar. In the interpreter, the weighted strategies generate no binding requests and the non-weighted strategies generate binding requests of the same magnitude as the number of suspensions generated; that is, the suspensions were caused by distributed *add/3* goals suspending remotely.

Figure 7.8 shows the number of binding requests generated for *fib*. The pattern of binding requests generated with the emulator is similar to the results with the interpreter. The non-weighted strategies generate some hundreds of binding requests, the same numbers as suspensions, showing that these are remote suspensions. The weighted strategies generate negligible binding requests since the vast majority (literally 99.9%) are local suspensions. A few binding requests are generated, however, for 16 processors or more, when none were generated in the equivalent interpreter experiments. This is because of a difference in operation between the interpreter and emulator. The interpreter does not allow any zero weighted goals to be exported to another processor. There is a situation, however, in which the emulator may export zero weighted goals. If a zero weighted goal suspends but is awakened while at the same time a non-zero weighted goal is being executed, then the zero weighted goal may get into the goal queue ahead of the non-zero weighted goal. This is because when a

time slice happens, the current goal (non-zero weighted) is pushed onto the front (local end) of the goal queue rather than being merged by weight. With the *fib* program, this may occasionally result in an *add/3* goal being scheduled after a *fib/2* goal, which is then subsequently exported, suspending remotely. This 'feature' could be removed by merging the current goal into the goal queue on a time slice but this would take more time.

In the case of fib, this 'feature' is invoked very rarely (an average of 10 times) and is unlikely to significantly affect the results.

To summarise the results of the experiments with *fib*:

- With small numbers of processors, the non-weighted strategies perform better than the weighted strategies. This is because the weighted strategies generate large numbers of local suspensions whereas the non-weighted strategies generate few remote suspensions.
- With larger numbers of processors (>25), the weighted strategies appear to be performing better than the non-weighted strategies. This is because the non-weighted strategies cause *add/3* goals to be stolen by idle processors, which immediately suspend on a remote variable, causing the processor to be idle again. With the weighted strategies idle processors steal *fib/2* goals which do not suspend immediately and may generate reasonable amounts of work.
- The AP strategies have better load balancing characteristics than the NN strategies.
- The weighted version of a distribution strategy has a better load balancing characteristic than the corresponding non-weighted strategy.
- Comparing the results from the interpreter and emulator, the speedup difference between the weighted and non-weighted strategies in the emulator is more prominent than in the interpreter. This is due to the reduction in average execution time which gives suspensions more effect on the execution.
- Compared with the results from the interpreter, the speedup characteristics of the emulator strategies are significantly worse.
- The load balancing characteristics for the AP strategies are worse for the emulator compared with the interpreter, but for the NN strategies load balance is the same.
- The suspensions and binding request characteristics are the same for the emulator and interpreter (ignoring the very small effect of some binding requests generated in the non-weighted strategies for the emulator where no binding requests are generated in the corresponding interpreter results).

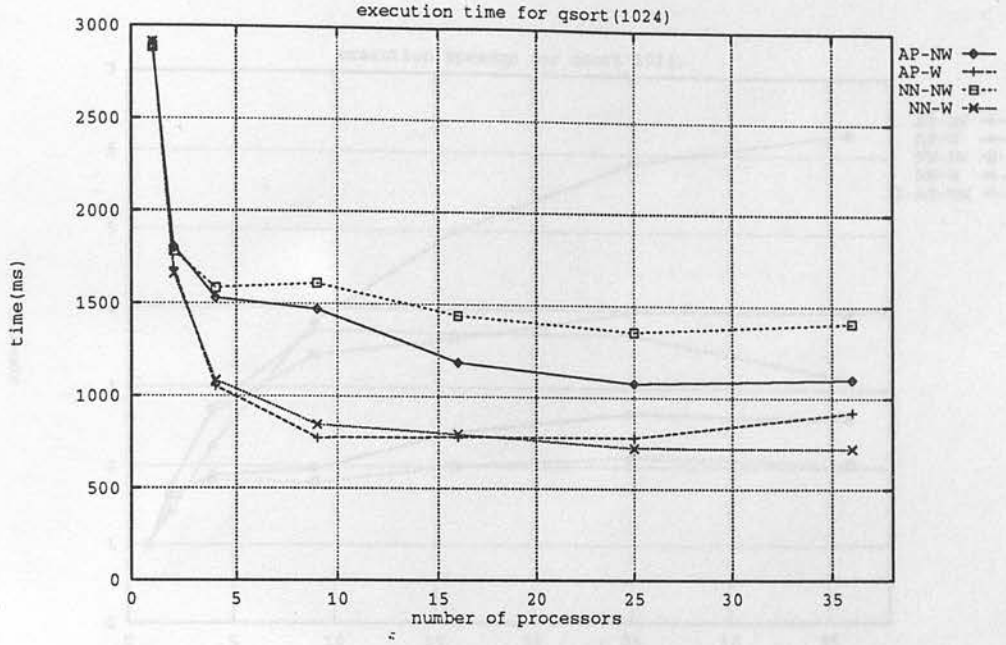

Figure 7.9: Emulator: Execution time for *qsort(1024)*

7.3.3 qsort

Figure 7.9 shows the execution time characteristic for *qsort*. The distribution strategies divide into two groups, the weighted strategies perform better than the non-weighted strategies. Within the groups, the AP-NW performs better than NN-NW, but NN-W performs better than AP-W. Overall, NN-W gives the least execution time.

Compared with the interpreter, executing *qsort* on the emulator is considerably faster; using 36 processors the interpreter achieves the same execution time as the emulator using 1 processor.

This result is reflected in the speedup graph, figure 7.10. NN-W gives the best speedup, reaching a peak performance of 4 times speedup at 25 processors. The best strategy for the interpreter experiments was AP-NW which achieved a better speedup of about 6 for 36 processors, although neither the interpreter nor the emulator result is impressive.

Figure 7.11 is the load balance plot for *qsort*. The graph is very similar to the results obtained for the interpreter: the same shape and magnitude, and the strategies are ranked in the same order. AP-NW has the best load balancing characteristic.

Figure 7.12 shows the suspensions characteristics for *qsort*. The strategies divide into the same two groups as was observed in the time and speedup graphs: the weighted strategies generate less suspensions than the non-weighted strategies. The strategies

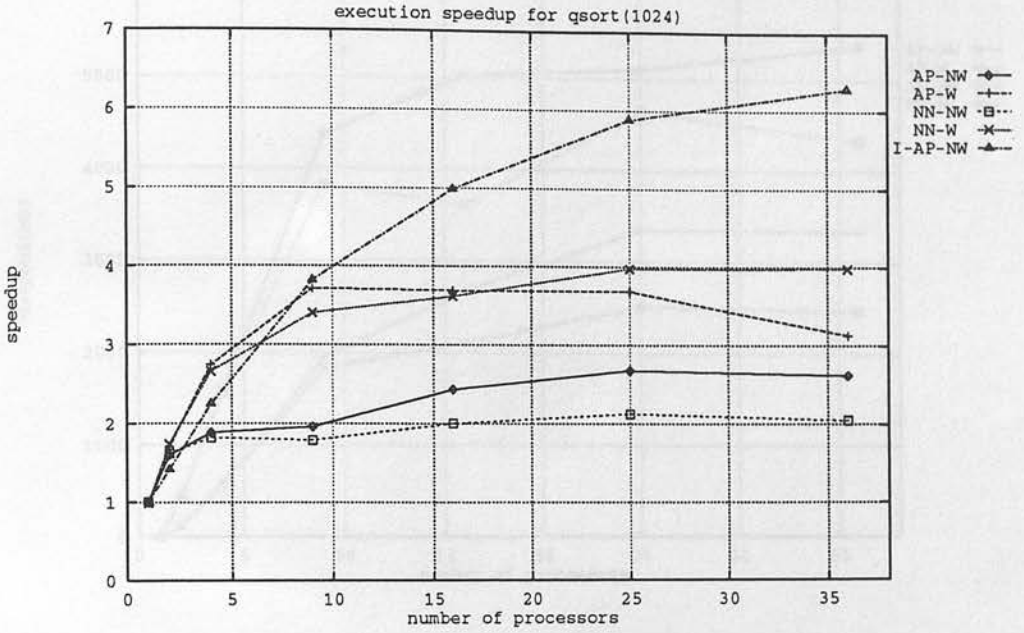

Figure 7.10: Emulator: Speedup for *qsort(1024)*

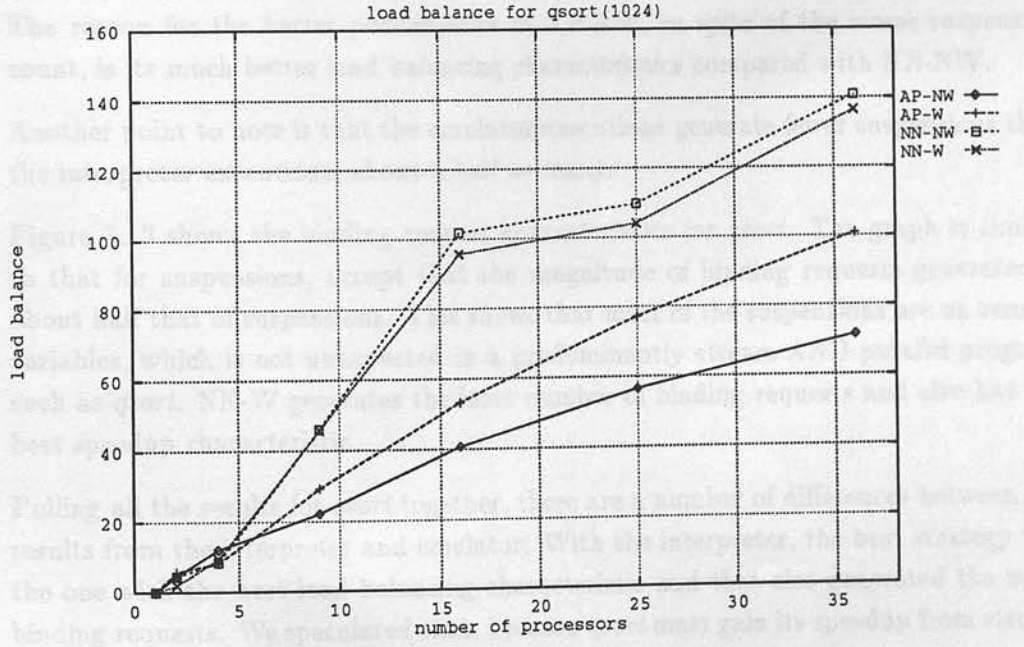

Figure 7.11: Emulator: Load balance for *qsort(1024)*

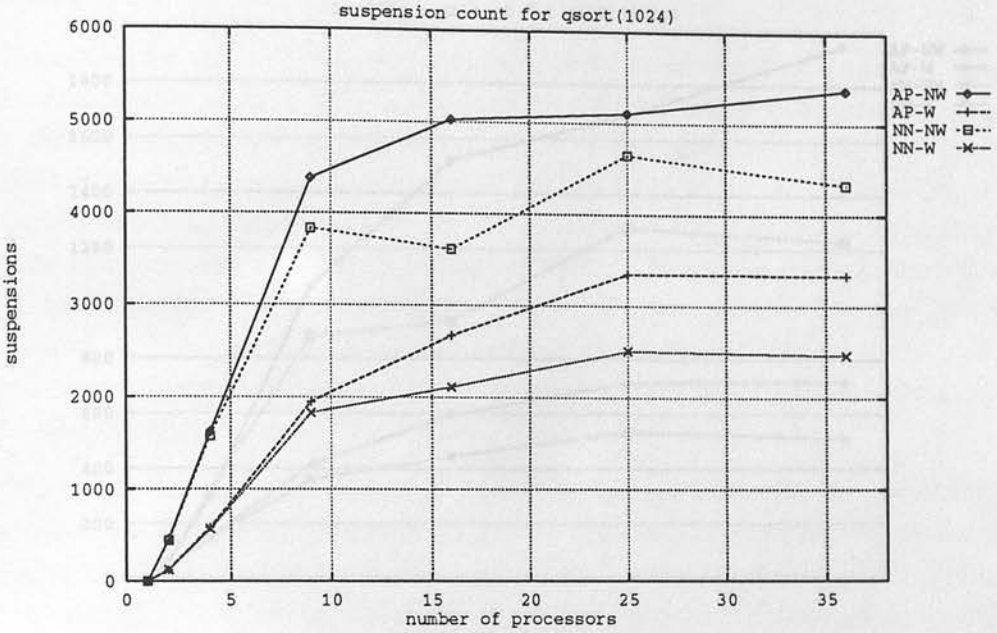

Figure 7.12: Emulator: Suspensions for *qsort(1024)*

are ranked by suspensions in almost the same order as speedup, with NN-W having the fewest suspensions and the best speedup. The main difference is that NN-NW generates less suspensions than AP-NW, but that AP-NW has a better speedup characteristic. The reason for the better performance of AP-NW, in spite of the worse suspension count, is its much better load balancing characteristics compared with NN-NW.

Another point to note is that the emulator executions generate fewer suspensions than the interpreter executions, about a half as many.

Figure 7.13 shows the binding request characteristics for *qsort*. The graph is similar to that for suspensions, except that the magnitude of binding requests generated is about half that of suspensions. This shows that most of the suspensions are on remote variables, which is not unexpected in a predominantly stream AND-parallel program such as *qsort*. NN-W generates the least number of binding requests and also has the best speedup characteristic.

Pulling all the results for *qsort* together, there are a number of differences between the results from the interpreter and emulator. With the interpreter, the best strategy was the one with the best load balancing characteristic and that also generated the most binding requests. We speculated that, because *qsort* must gain its speedup from stream AND-parallelism where producer and consumer goals execute on different processors, then a high binding request count might indicate a good speedup characteristic for this program.

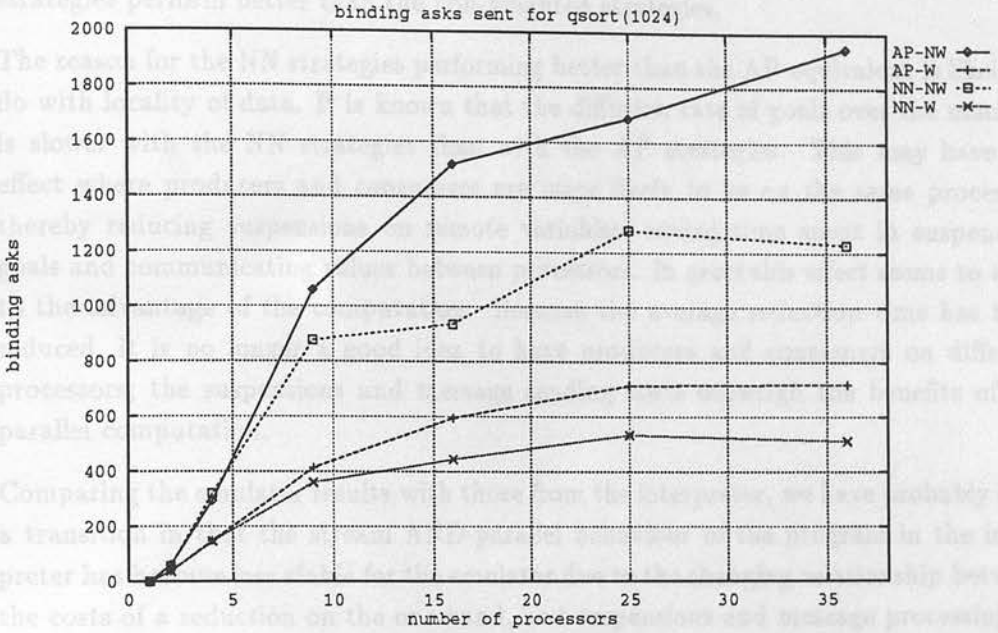

Figure 7.13: Emulator: Binding requests for *qsort(1024)*

But the results from the emulator do not support this view. The best strategy is the one that generates the least suspensions and binding requests, and has the third best (or second worst) load balancing characteristic. These results support the opposite view that generating suspensions and binding requests is costly and should be avoided. It seems that the lessening of the average reduction time has made suspensions and binding requests more prominent as factors against speedup.

This is a case where good load balance is not the ultimate indicator of good performance. Load balance is still important, however, since AP-NW generates more suspensions and binding requests than NN-NW but has a much better speedup characteristic, due to superior load balancing.

A final point to consider is: why does NN-W perform better than the other strategies? This is a rare occasion where a nearest-neighbours or weighted strategy performs better than the other strategies; in this case both attributes are present. It does not have good load balancing but it does have very low suspension and binding request counts. Choosing a weighted strategy makes good sense; it means that *part/4* goals will be executed locally first rather than being sent out for remote execution. This is good because *part/4* just iterates down a list of integers, as does packing it for transmission, which means that packing and sending a *part/4* goal is likely to take as much time as executing it in the first place. Keeping a *part/4* goal, instead of distributing it, will save time and reduce suspensions. This is probably the main reason why the weighted

strategies perform better than the non-weighted strategies.

The reason for the NN strategies performing better than the AP equivalent is likely to do with locality of data. It is known that the diffusion rate of goals over the machine is slower with the NN strategies than with the AP strategies. This may have the effect where producers and consumers are more likely to be on the same processor, thereby reducing suspensions on remote variables, saving time spent in suspending goals and communicating values between processors. In *qsort* this effect seems to work to the advantage of the computation. Because the average reduction time has been reduced, it is no longer a good idea to have producers and consumers on different processors; the suspensions and message sending costs outweigh the benefits of the parallel computation.

Comparing the emulator results with those from the interpreter, we have probably seen a transition in that the stream AND-parallel behaviour of the program in the interpreter has become less viable for the emulator due to the changing relationship between the costs of a reduction on the one hand, and suspensions and message processing on the other hand.

To summarise the results from *qsort*:

- The weighted strategies show the best speedup, with NN-W having the best performance.
- Suspension counts and the number of binding requests processed is a better indicator of performance than load balance. NN-W generates the least number of suspensions and binding requests and has the best performance.
- The bad load balancing characteristic of NN-W probably helps to keep goals and data on the same processor, reducing suspensions and binding requests, and improving performance. This combined with the weighting, which does not allow *part/4* goals to be distributed, is probably the reason why this strategy is the best performer.
- The results for the emulator are the opposite of those for the interpreter. In the interpreter, the strategy with the best load balance and the 'worst' binding request characteristic has the best performance (AP-NW). But with the emulator, the strategy with the worst load balancing and best binding request characteristic has the best performance.

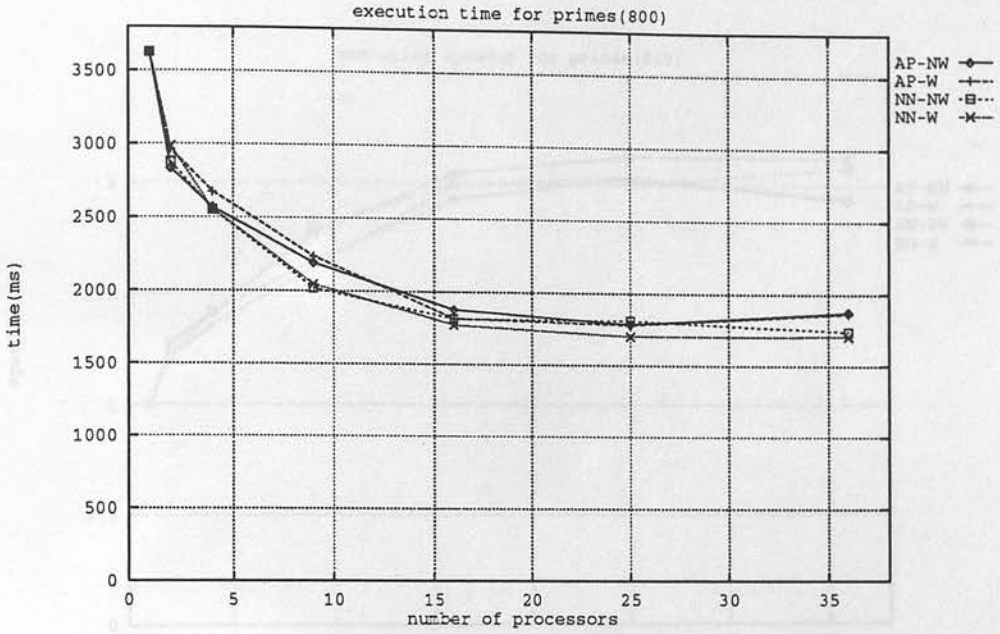

Figure 7.14: Emulator: Execution time for *primes(800)*

7.3.4 primes

Figure 7.14 shows the execution time characteristic for *primes*. There is not a large difference between the different goal distribution strategies, but the NN strategies perform slightly better than the AP strategies, with NN-W the best performer. A point to note is that the best execution time for an interpreter execution is about 4.5 s on 36 processors compared with between 3.5 s (1 processor) and 1.7 s (36 processors) for the emulator.

Figure 7.15 shows the speedup characteristic for *primes*. The speedup characteristic is not good in that the best strategy, NN-W, attains a speedup of just over 2 at 25 processors. The other strategies peak at a value of 2 at 25 processors; after 25 processors the AP strategies have worse speedup, but NN-NW performs a little better.

The speedup figure of about 2 in the emulator compares with a speedup of close to 7 attained with the interpreter. There is an obvious trend throughout the benchmark programs that the improvement in average reduction time has made it harder to obtain speedup with the emulator.

Figure 7.16 shows the load balancing characteristic for *primes*. The strategies split into the usual groups of the AP strategies having better load balancing characteristics than the NN strategies. There is no difference in load balancing between weighted and non-weighted versions of the AP strategies. The weighted version of the NN strategies

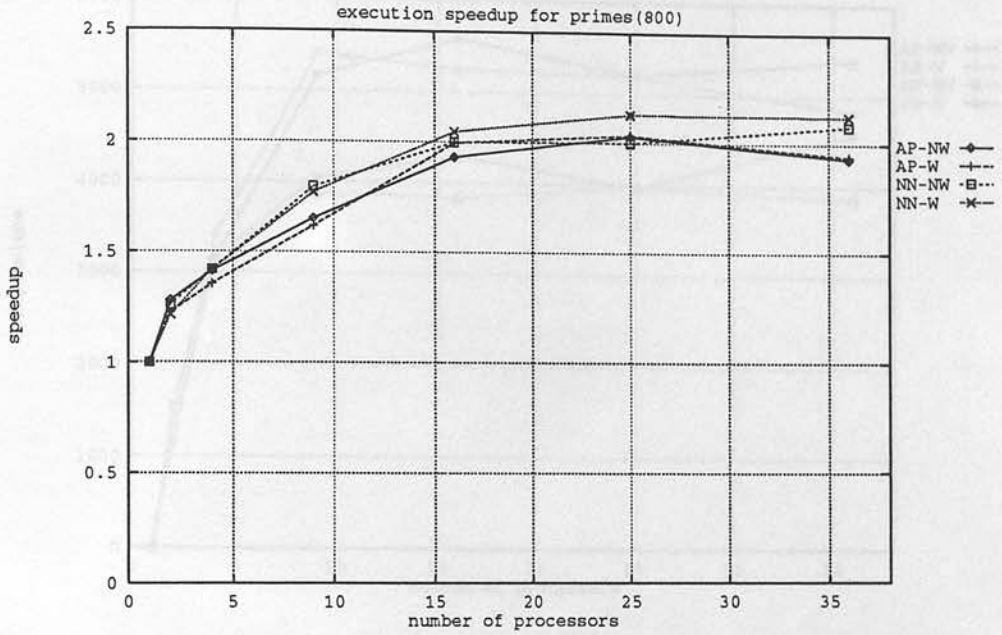

Figure 7.15: Emulator: Speedup for *primes(800)*

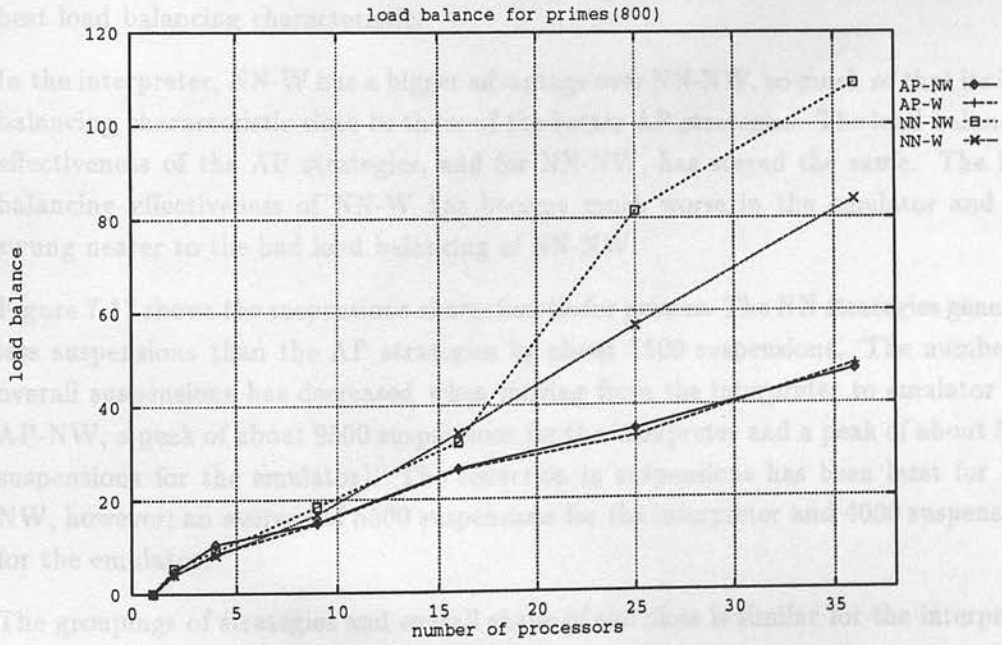

Figure 7.16: Emulator: Load balance for *primes(800)*

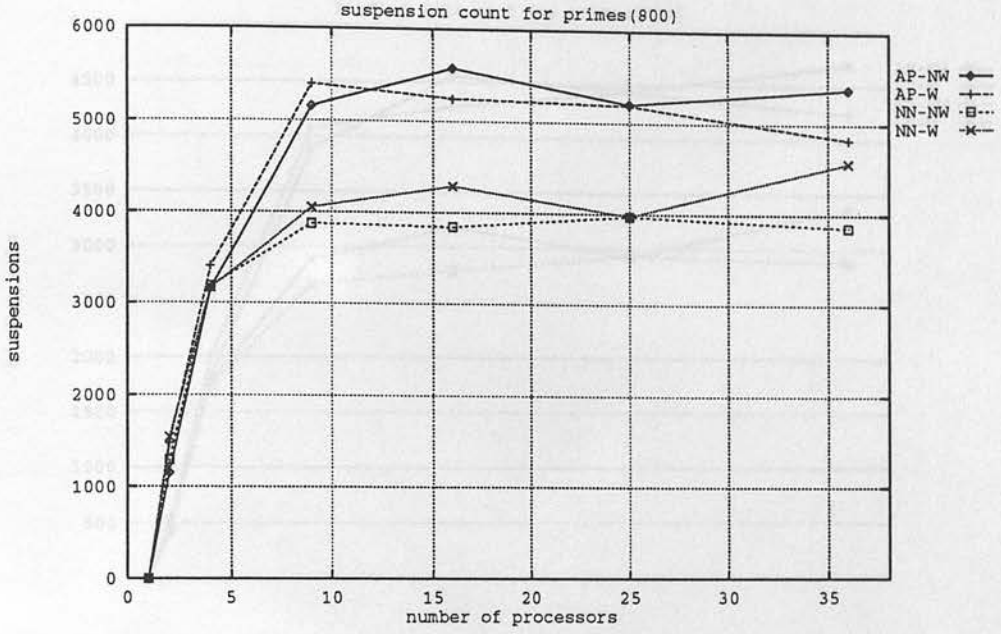

Figure 7.17: Emulator: Suspensions for *primes(800)*

does have an advantage over the non-weighted strategy.

Here again is a case where the best performing strategy (NN-W) does not have the best load balancing characteristic.

In the interpreter, NN-W has a bigger advantage over NN-NW, so much so that its load balancing characteristic close to those of the better AP strategies. The load balancing effectiveness of the AP strategies, and for NN-NW, has stayed the same. The load balancing effectiveness of NN-W has become much worse in the emulator and has swung nearer to the bad load balancing of NN-NW.

Figure 7.17 shows the suspensions characteristic for *primes*. The NN strategies generate less suspensions than the AP strategies by about 1500 suspensions. The number of overall suspensions has decreased when moving from the interpreter to emulator (for AP-NW, a peak of about 9500 suspensions for the interpreter and a peak of about 5500 suspensions for the emulator). The reduction in suspensions has been least for NN-NW, however; an average of 5500 suspensions for the interpreter and 4000 suspensions for the emulator.

The groupings of strategies and overall shape of the plots is similar for the interpreter and emulator.

Figure 7.18 shows the binding requests generated for *primes*. This shows the same grouping of strategies for the suspensions graph (NN strategies generate less binding

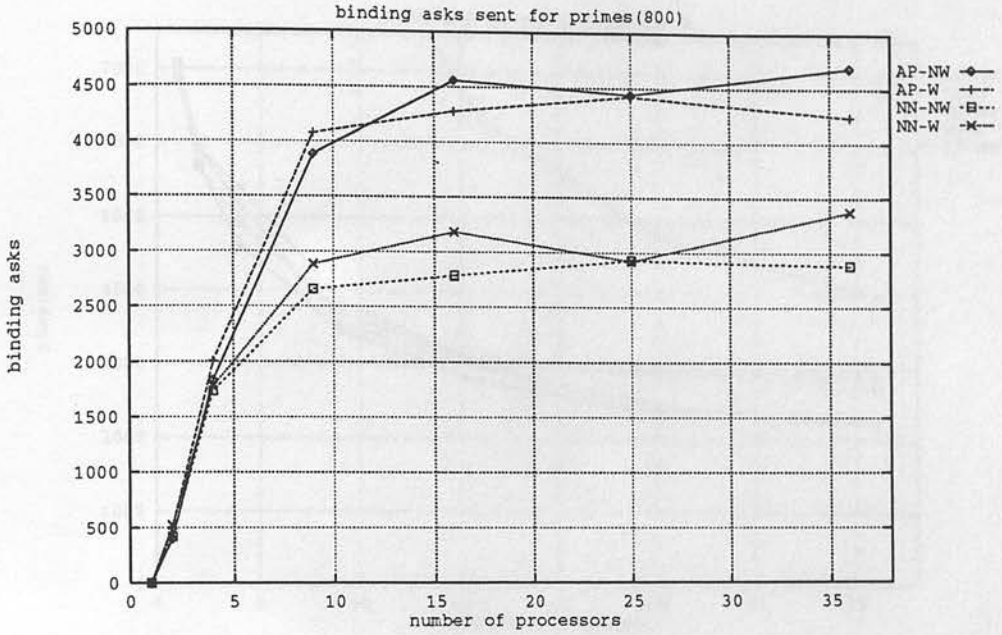

Figure 7.18: Emulator: Binding requests for *primes(800)*

requests than the AP strategies), and both graphs have the same shape indicating that most of the suspensions are on remote variables. The NN strategies do, however, generate proportionally less binding requests than they do suspensions compared to the AP strategies.

These results point to the fact that fewer goals are distributed to processors with the NN strategies than with the AP strategies, and that this reduces suspensions and binding requests. This reduction in suspensions and messages would seem to compensate for the worse load balancing characteristics to such an extent that NN-W performs slightly better than the other strategies.

To summarise results for *primes*:

- All the strategies execute in roughly the same time and have similar speedup characteristics, a very poor speedup of about 2. If one had to choose the best performing strategy then NN-W has a slightly better performance than the other strategies.
- The load balancing characteristics of the AP strategies are much better than for those of the NN strategies.
- The suspension and binding request counts are lower for the NN strategies than for the AP strategies.

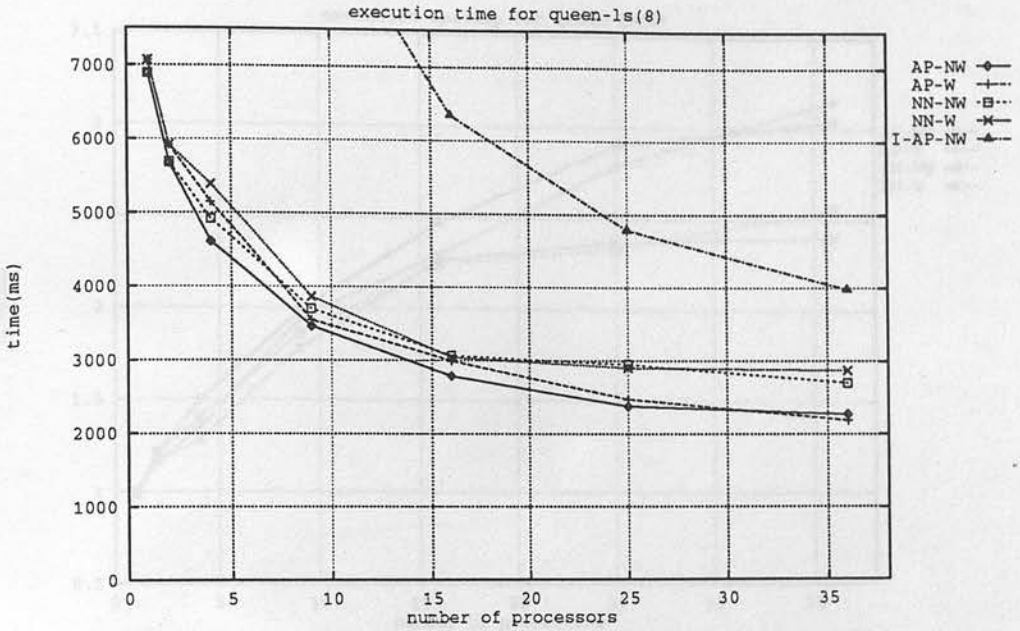

Figure 7.19: Emulator: Execution time for *queen-ls(8)*

- It would seem that the effects of load balancing and suspensions/messages cancel each other out such that the execution times for the strategies are similar.

7.3.5 queen-ls

Figure 7.19 shows the execution time characteristic for *queen-ls*. For less than 25 processors there is little difference between the distribution strategies, although AP-NW is the best strategy in this range.

For 25 processors and over, the AP strategies are clearly better than the NN strategies. There is not enough difference between the weighted and non-weighted versions of strategies to be sure which version is better.

Also plotted on the graph is the execution time characteristic of the interpreter executing AP-NW (keyed as I-AP-NW). This shows that the interpreter with 36 processors executes at about the same speed as the emulator with 9 processors. At 36 processors the emulator, using the AP strategies, is about twice as fast as the interpreter.

Figure 7.20 shows the speedup characteristics for *queen-ls*. The AP strategies reach a speedup of just over 3 for 36 processors while the NN strategies reach a speedup of 2.5. The speedup characteristics for the emulator are similar in shape to those for the in-

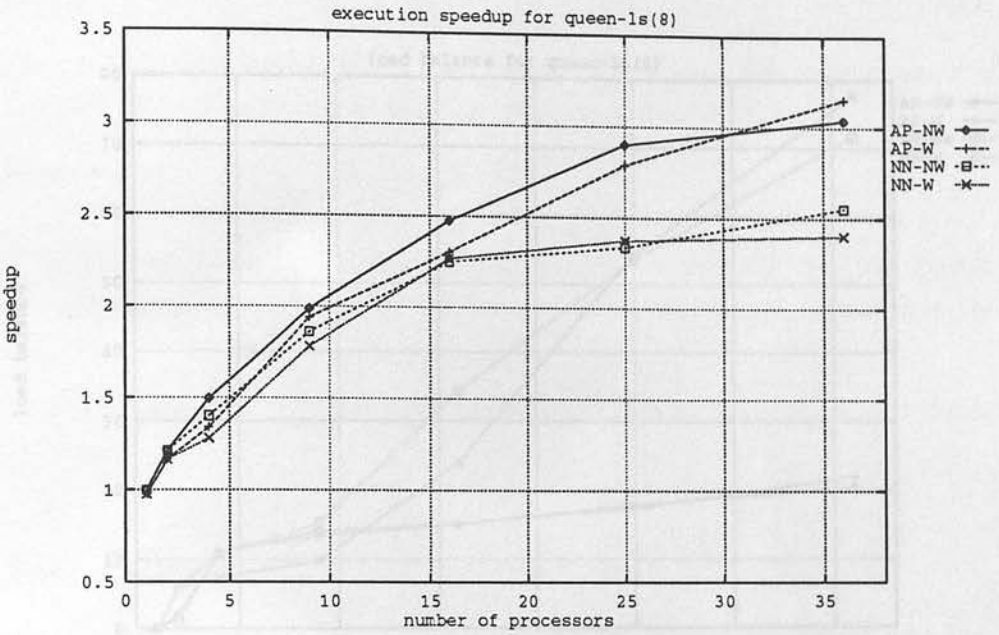

Figure 7.20: Emulator: Speedup for *queen-ls(8)*

terpreter, the major difference is that the best interpreter strategies reach a speedup of about 11 for 36 processors (as compared to 3 for the emulator). Again an improvement in average reduction time has led to an improvement in execution time but also a degradation in speedup characteristic.

Figure 7.21 shows the load balancing characteristic for *queen-ls*. This is very similar to the equivalent characteristic from the interpreter results. The emulator NN strategies have slightly worse load balancing properties when compared with the interpreter versions; the emulator AP strategies have significantly worse load balance values when compared with the interpreter versions. But still the AP strategies have vastly superior load balancing properties than the NN strategies. There is nothing to choose between weighted or non-weighted versions of a strategy.

Figure 7.22 shows the suspension characteristic for *queen-ls*. The AP strategies generate less suspensions than the NN strategies, although this difference has narrowed to nothing by 36 processors. There is little difference in suspensions between weighted and non-weighted strategies.

When compared with the interpreter, the emulator AP strategies have not changed a great deal. Both graphs have a similar shape and similar numbers of suspensions are generated (about 8000–9000 suspensions). With the emulator NN strategies, however, the number of suspensions has reduced slightly from around 11,000, to 9000. But generally the suspensions characteristic for the emulator and the interpreter are very

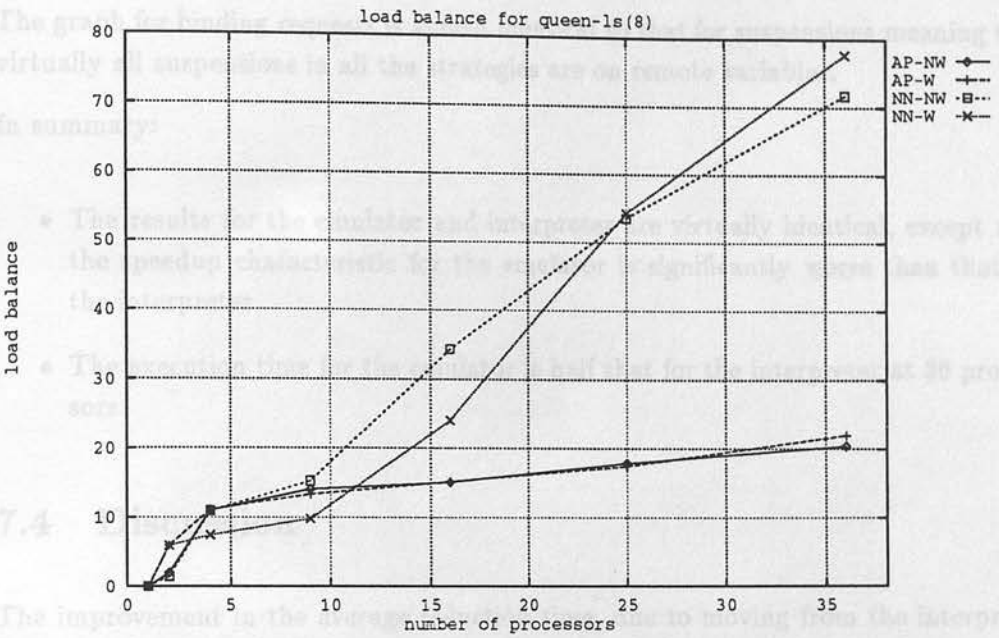

Figure 7.21: Emulator: Load balance for *queen-ls(8)*

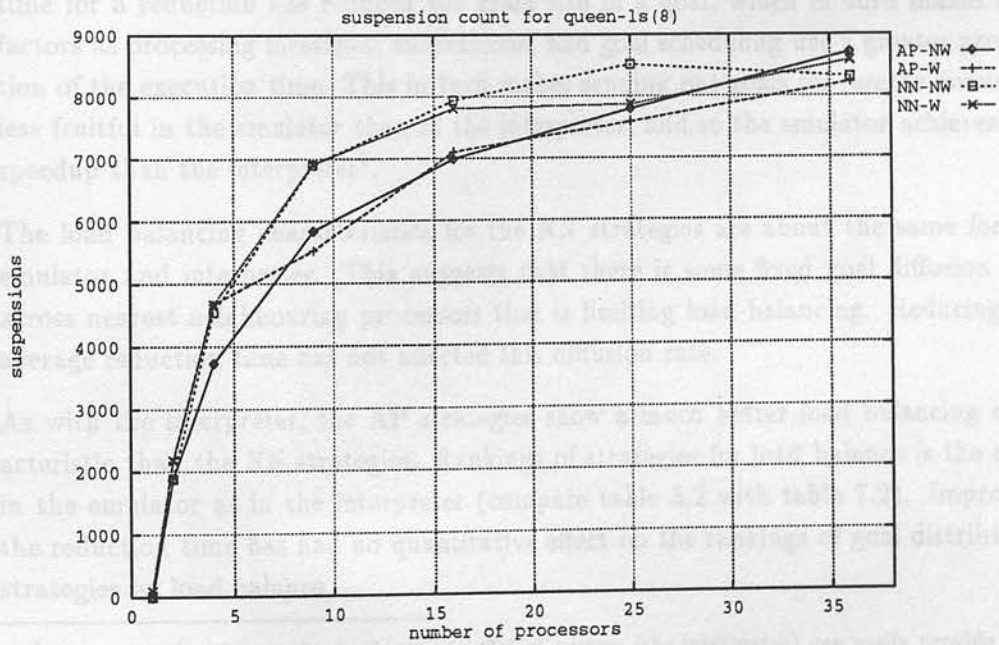

Figure 7.22: Emulator: Suspensions for *queen-ls(8)*

similar.

The graph for binding requests is almost identical to that for suspensions meaning that virtually all suspensions in all the strategies are on remote variables.

In summary:

- The results for the emulator and interpreter are virtually identical, except that the speedup characteristic for the emulator is significantly worse than that for the interpreter.
- The execution time for the emulator is half that for the interpreter at 36 processors.

7.4 Discussion

The improvement in the average reduction time, due to moving from the interpreter to the emulator system, has had one obvious effect: the execution time has reduced significantly for all the test programs.

A negative effect has been, however, that the emulator does not attain the same levels of speedup as the interpreter. This was to be expected, because improving the average time for a reduction has reduced the grain size of a goal, which in turn makes such factors as processing messages, suspensions, and goal scheduling use a greater proportion of the execution time. This in turn makes sending out goals for remote execution less fruitful in the emulator than in the interpreter, and so the emulator achieves less speedup than the interpreter[1].

The load balancing characteristics for the NN strategies are about the same for the emulator and interpreter. This suggests that there is some fixed goal diffusion rate across nearest neighbouring processors that is limiting load balancing. Reducing the average reduction time has not affected this diffusion rate.

As with the interpreter, the AP strategies show a much better load balancing characteristic than the NN strategies. Rankings of strategies for load balance is the same in the emulator as in the interpreter (compare table 5.2 with table 7.2). Improving the reduction time has had no quantitative effect on the rankings of goal distribution strategies by load balance.

[1] One way of looking at this is that an inefficient system (the interpreter) can easily provide good speedups compared with an efficient system (the emulator) because it has an 'inflated' average reduction time.

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	AP-NW	-NW	NN-W	NN	AP-W
2nd	AP-W	-NW	AP-W	NN	AP-NW
3rd	NN-NW	-W	AP-NW	AP	NN
4th	NN-W	-W	NN-NW	AP	NN

Table 7.1: Ranking of schedulers by speedup

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	AP	AP-W	AP-NW	AP	AP
2nd	AP	AP-NW	AP-W	AP	AP
3rd	NN	NN-W	NN	NN-W	NN
4th	NN	NN-NW	NN	NN-NW	NN

Table 7.2: Ranking of schedulers by load balance

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	n/r	-NW	NN-W	NN	n/r
2nd	n/r	-NW	AP-W	NN	n/r
3rd	n/r	-W	NN-NW	AP	n/r
4th	n/r	-W	AP-NW	AP	n/r

Table 7.3: Ranking of schedulers by suspensions

	hanoi	*fib*	*qsort*	*primes*	*queen-ls*
1st	n/r	-W	NN-W	NN	n/r
2nd	n/r	-W	AP-W	NN	n/r
3rd	n/r	-NW	NN-NW	AP	n/r
4th	n/r	-NW	AP-NW	AP	n/r

Table 7.4: Ranking of schedulers by binding requests

For many programs, suspensions and binding request characteristics are a better indication of speedup than load balancing (see tables 7.1–7.4). This is because the proportion of time spent in suspending goals and processing messages has become more prominent (due to lessening of the prominence of time spent reducing) in the emulator than in the interpreter. Reducing the number of suspensions and binding requests has an effect on execution time in the emulator, whereas it seems to be largely irrelevant in the interpreter.

For the first time NN strategies have been observed to perform better than AP strategies (in *fib*, *qsort*, and *primes*). This may be because of the slower diffusion rate of goals in the NN strategies, keeping more goals on the same processor, reducing the number of suspensions and binding requests. Although this effect limits parallelism, much of the parallelism available to the system is useless because it would take more time to exploit it than would be saved through utilising it.

Using weights can also produce improved speedups (in *qsort* and *queen-ls*) but the effect is not very great.

A significant effect that has not been remarked upon so far is that for *qsort* and *primes*, not only do they not show good speedups, but that also the maximum speedup is achieved with few processors; increasing the number of processors after that point gives no speedup or can give slow down (the AP strategies in figure 7.15). Although this is due to an extent to the nature of the programs and the reduction in average time of a reduction, it may also be because the computation is not large enough to sustain work over the processors for very long. When reducing the average reduction time, one way of possibly 'compensating' is to increase the problem size, by for example, increasing the number of primes generated or the size of the list sorted. The nature of the algorithms, however, would mean that the size of the computation would need to be increased substantially. The speedup attainable for *qsort* is proportional to the depth of the tree of *part/4* goals generated, which is $\log N$ where N is the number of elements in the list (for a balanced sort). Doubling the length of the list will only increase the speedup by 1 at most. Increasing the problem size of *primes* is likely to have much less effect since most of the *filter/4* goals are suspended most of the time.

We have elected not to increase the problem size but to keep all factors the same except for the average time for a reduction. Examining the effects of scaling the problem size is left for further work.

7.5 Summary

The results of executing the test programs, with the various distribution strategies, on the emulator were presented, and they were compared with the results from the similar experiments with the interpreter described previously.

The following conclusions were drawn:

- The execution time of programs executing under the emulator are greatly reduced compared to when using the interpreter. This is due to the substantially smaller average time for a reduction in the emulator.
- A reduction in execution time has also resulted in a reduction in effectiveness in speedup; the programs running under the emulator show a worse speedup characteristic than when running under the interpreter.
- Load balancing has become harder for the AP strategies when using the emulator compared to using the interpreter. For the NN strategies, load balancing is similar regardless of the system used. This suggests that the rate of diffusion of goals between nearest neighbour processors is limiting load balance, and this rate remains unaffected by average reduction time.
- The number of suspensions and binding requests has become more prominent due to the smaller average time for a reduction in the emulator (compared to the interpreter). This has made levels of suspensions and binding requests a better indicator of good speedup than load balancing.
- Consequently, a **low** suspension and binding request rate points to a **good** speedup for the emulator for all programs. This is contrary to the results for the interpreter where is seems that, for predominantly stream AND-parallel programs, a **high** suspension and reduction count pointed to **good** speedup.
- Although using weights can improve performance, that improvement is very small.
- For some programs, a NN strategy has the best performance. This may be due to the slow effects of goals diffusing between processors tending to increase locality, keeping goals that would be inefficient to execute remotely on the local processor.
- There is no overall best goal distribution strategy.

Chapter 8

Other experiments and further work

8.1 Introduction

In this chapter we present two types of information:

- ideas for which we have made some preliminary experiments;
- ideas for further work.

More specifically, the chapter begins by presenting experiments for a modification to the goal distribution strategies we have so far investigated. The modification is to send goals that suspend on a single remote variable to the processor on which that remote variable resides. This modification was introduced to see if the number of suspensions would reduce and improve execution times.

Some experiments using larger numbers of processors than have been used so far are presented. The motivation behind these experiments is to see how the goal distribution strategies would perform with very large numbers of processors, to see, for example, if execution performance degrades due to load balancing and message congestion problems. Preliminary experiments have been made using these ideas and the results are presented.

The final section consists of ideas for furthering the work presented in this thesis.

8.2 Goals migrate to data

An observation made during our experiments is that, for our test programs, the majority of goals that suspend do so on just one variable.

In the current system implementations, when a goal suspends on a variable, the variable is checked to see if it is a local or remote variable. If local then the goal is added to the suspension list of the local variable. If remote, then a request for the binding of the remote variable is sent out, an entry is made for it in the Remote Binding Array, and the suspending goal is added to a local suspension list referenced from the Remote Binding Array Entry. If a binding for a remote variable is subsequently received, then the binding is made locally in the Remote Binding Array and any goals in a suspension list referenced by the Remote Binding Array entry are awakened for execution.

If a goal suspends on one variable only then an alternative course of action could be taken; instead of asking the remote processor for a copy of the variable binding, the system could send the goal to the remote processor. In this way a suspension on the local processor is exchanged for sending a binding request to the remote processor. One way of looking at this new mechanism is to picture the goal moving to where the data is rather than the data moving to the goal. This strategy will be refered to as the **goals-to-data** strategy.

When the remote processor receives the goal, it is treated as any other goal; it is scheduled on the goal queue.

The hope with this mechanism is that time is saved if goals that might suspend are sent to where the data they need is stored: sending a goal may be quicker than suspending it.

It should be noted that if a goal suspends on more than one variable, since it suspends on different variables from different clauses in a relation, then the goal is suspended locally in the usual manner. This is because the goal cannot be sent to more than one processor and since a choice cannot be made between the two places to send the goal is suspended locally[1].

With the conventional mechanism, a goal is suspended, then a message is sent to the remote processor, a message is received from the remote processor, and then the goal awakened. This gives a tally of 2 messages processed and a goal suspended and

[1]This is not strictly true. If a goal suspends on a number of variables then as long as those variables all reside on the same processor then the goal can be sent to that processor for further consideration. Problems only arise when there is more than one processor to choose from.

However, testing to see if the different variables all reside on the same processor is time consuming and may be rare. Testing to see if there is only one variable on which the goal has suspended is simple and may be more common.

awakened.

With the 'goals-to-data' mechanism it might be that a suspending goal is sent to the remote processor, is then scheduled, and if there is enough data then the goal will complete to termination. This gives a work tally of just sending 1 message.

This, of course, is the ideal case. There are many other scenarios that are likely to happen. One case is that the suspending goal is sent and scheduled on the remote processor, and then suspends locally because of insufficient data.

The worst case is where a goal continuously follows a chain of remote variable references from one processor to another. For example, a goal is on the point of suspending on remote variable A and so it is sent to the processor on which A resides. Upon arrival, the goal is scheduled for execution but on execution it is found that variable A is bound to a further remote variable B. The goal is sent to the processor on which B resides, is scheduled for execution, but B happens to be bound to another remote variable C. In this way it would be possible for the goal to chase along a long chain of variable references across processors. It is easy to see that this would be very expensive since the goal is being continually (re)transmitted and execution of it is attempted at each stage.

Considering all these scenarios, it is expected that the goals-to-data scheduling technique will only give benefits if more often than not the remote processor has the data needed for the suspending goal to continue execution.

8.2.1 Experiments and results

The experiments were carried out in a similar manner as for those reported in previous chapters.

The difference with these experiments was that the goals-to-data feature was tried with the AP strategies. That is, the results reported here are for AP-NW, AP-W, G2D-NW, and G2D-W.

G2D-NW stands for goals-to-data all-processors no weights; G2D-W stands for goals-to-data all-processors with weights.

8.2.1.1 hanoi

The *hanoi* program is a control program in that the goals-to-data modification is not expected to make any difference to program execution. This is because *hanoi* generates no suspensions and so the goals-to-data option should not be invoked.

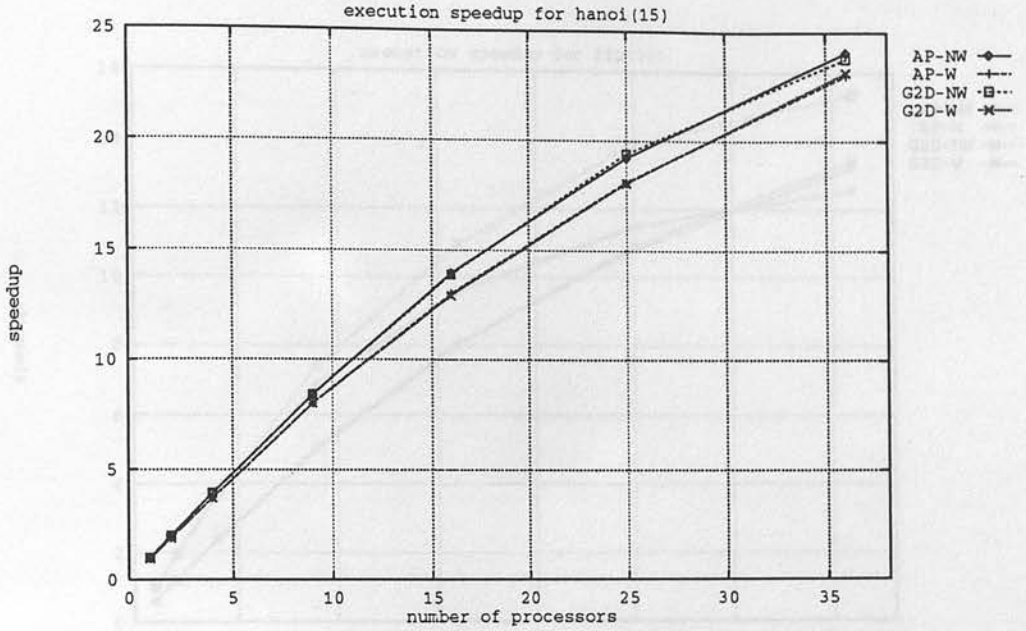

Figure 8.1: Goals-to-data: Speedup for *hanoi(15)*

Figure 8.1 is the speedup characteristic for *hanoi*. It shows that there is indeed no difference between having the goals-to-data option for either weighted or non-weighted strategies.

All other measures show the same trend. The control program behaves as was expected.

8.2.1.2 fib

Figure 8.2 shows the speedup characteristic for *fib*.

The plot for G2D-NW is an improvement over AP-NW: at 36 processors, G2D-NW has a speedup of over 15, and AP-NW has a speedup of just over 12. There is no difference in speedup characteristic for the weighted version of the strategy.

The suspensions characteristic, presented in figure 8.3, shows that the G2D-AP-NW strategy produces about half the suspensions that the AP-NW strategy produces; that is, 370 suspensions against 620 suspensions at 36 processors.

There is no reduction in the suspension count for the weighted strategies. This is expected since for these strategies all the suspensions are local; the goals-to-data feature will only affect remote suspensions.

Figure 8.4 shows the binding requests characteristics for *fib*.

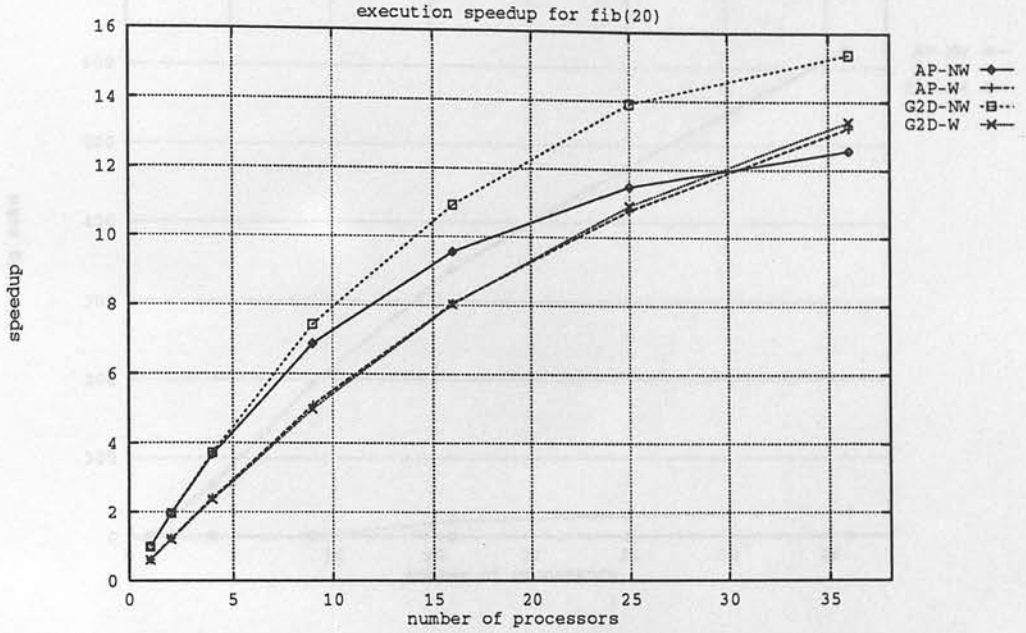

Figure 8.2: Goals-to-data: Speedup for *fib(20)*

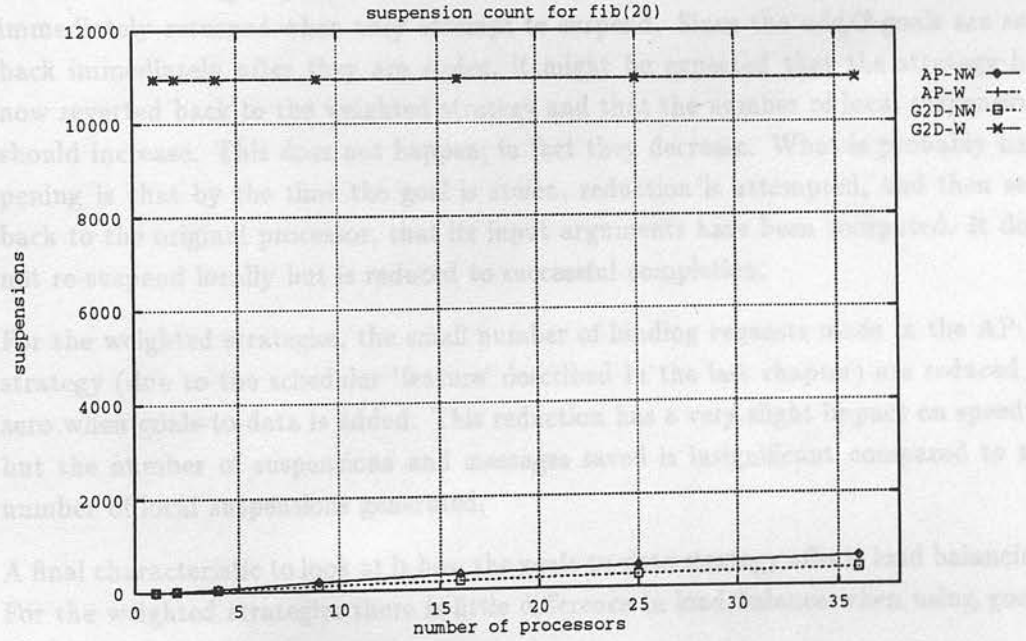

Figure 8.3: Goals-to-data: Suspensions for *fib(20)*

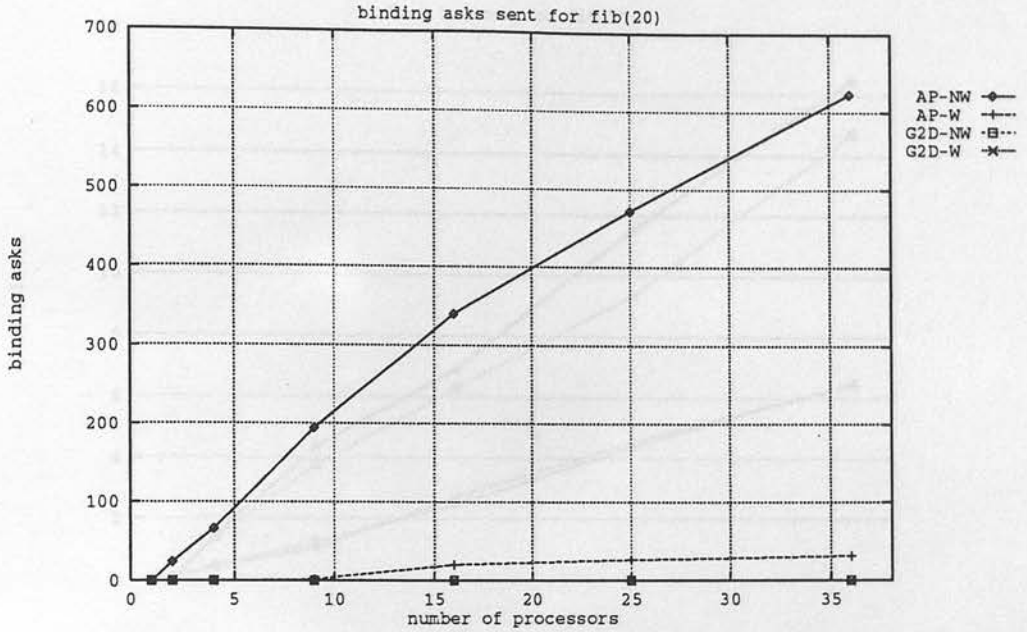

Figure 8.4: Goals-to-data: Binding requests for *fib(20)*

Here it is clear the effect that the goals-to-data feature is having on the goal distribution strategies. For the non-weighted strategies, using goals-to-data has reduced the number of binding requests to zero. The *add/3* goals stolen by idle processors are immediately returned when they attempt to suspend. Since the *add/3* goals are sent back immediately after they are stolen, it might be expected that the strategy has now reverted back to the weighted strategy and that the number of local suspensions should increase. This does not happen; in fact they decrease. What is probably happening is that by the time the goal is stolen, reduction is attempted, and then sent back to the original processor, that its input arguments have been computed. It does not re-suspend locally but is reduced to successful completion.

For the weighted strategies, the small number of binding requests made in the AP-W strategy (due to the scheduler 'feature' described in the last chapter) are reduced to zero when goals-to-data is added. This reduction has a very slight impact on speedup but the number of suspensions and messages saved is insignificant compared to the number of local suspensions generated.

A final characteristic to look at is how the goals-to-data strategy affects load balancing. For the weighted strategies there is little difference in load balance when using goals-to-data.

For the non-weighted strategies, using goals-to-data improves load balancing, although not by a great amount when compared with the load balancing characteristics of the

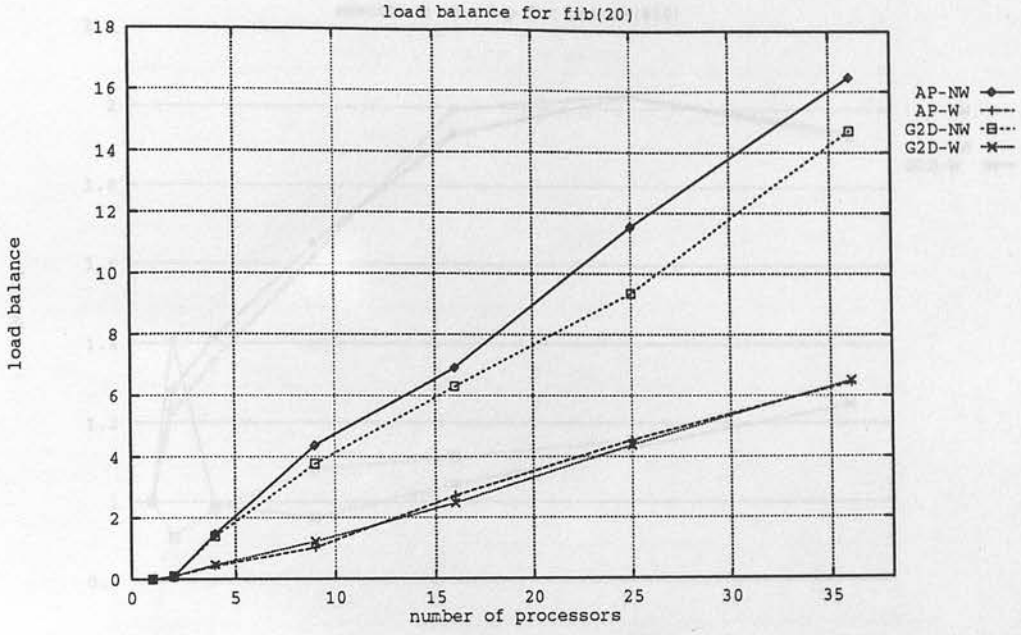

Figure 8.5: Goals-to-data: Load balancing for *fib(20)*

weighted strategies.

Drawing all these results together, we summarise:

- For the non-weighted strategies, the sending out of *add/3* goals to suspend remotely on another processor is very costly. Using goals-to-data removes the cost of the goal suspending on a remote variable but adds the alternative cost of sending the goal back. The cost of sending the goal back must be much smaller than the cost of suspending on a remote variable since G2D-NW shows a significant improvement in speedup over AP-NW.

- In a strategy that generates few suspensions on remote variables, as for the weighted strategies for *fib*, using goals-to-data will make little or no difference to performance.

8.2.1.3 primes

Performing the same experiments on the *primes* program does not give the same positive results as for *fib*.

Figure 8.6 is the graph of speedup for *primes(800)*. It shows that using the goals-to-data strategy has a detrimental effect on system performance: speedup is reduced from

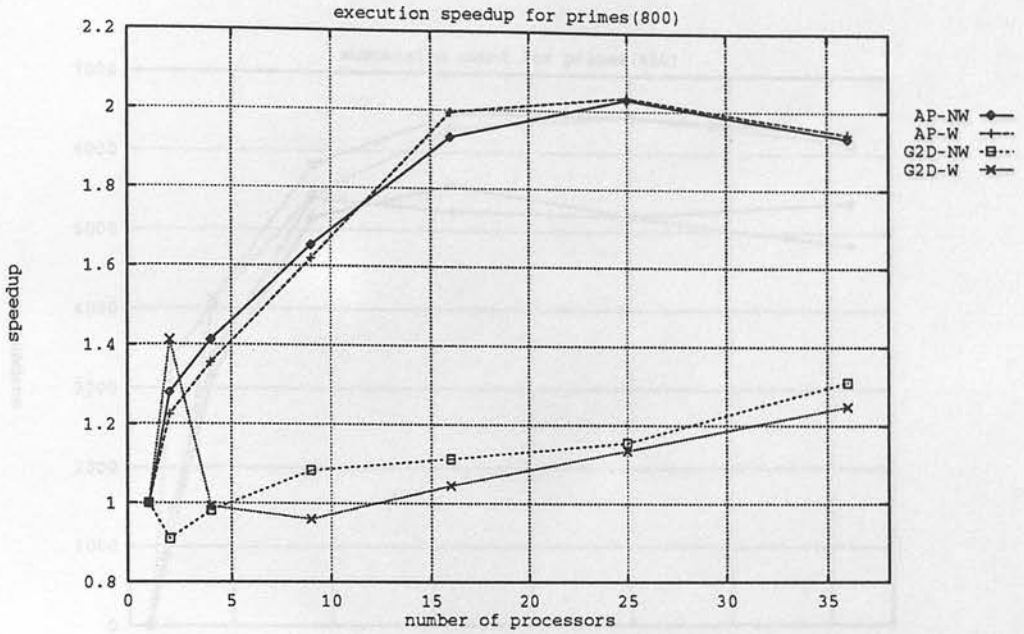

Figure 8.6: Goals-to-data: Speedup for *primes(800)*

2 to 1 for 36 processors. There is no difference in performance whether weights are used or not.

Figure 8.7 shows the suspensions characteristic for *primes*. Suspensions have increased, rather than decreased, when using the goals-to-data feature; the number of suspensions has increased from a peak of 5000 suspensions to 6500 suspensions.

Using goals-to-data does reduce the number of binding requests made to zero, as is shown in figure 8.8, for both weighted and non-weighted strategies. This shows that all the suspensions on remote variables are on just one remote variable. Without goals-to-data, the number of binding requests peaks at 4500 requests.

Although the number of binding requests has reduced, the number of suspensions has increased, when using goals-to-data. If when the suspending goal is returned (to the processor on which the variable resides) it were to suspend locally straight away, then the number of suspensions should not increase; remote suspensions should be converted to local suspensions.

This is not happening since the number of suspensions has increased. What is probably happening is that the order in which goals are executed is dramatically changed and more suspensions are generated as a result.

An obvious argument as to why the goals-to-data strategy is bad for *primes* is to consider how such a strategy behaves with a predominantly stream AND-parallel program.

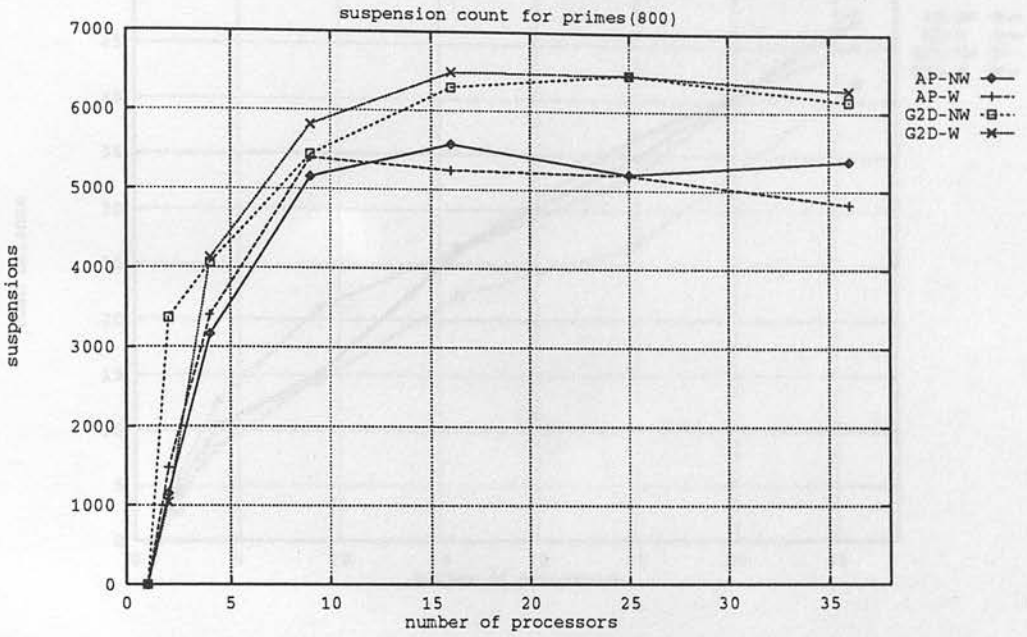

Figure 8.7: Goals-to-data: Suspension for *primes(800)*

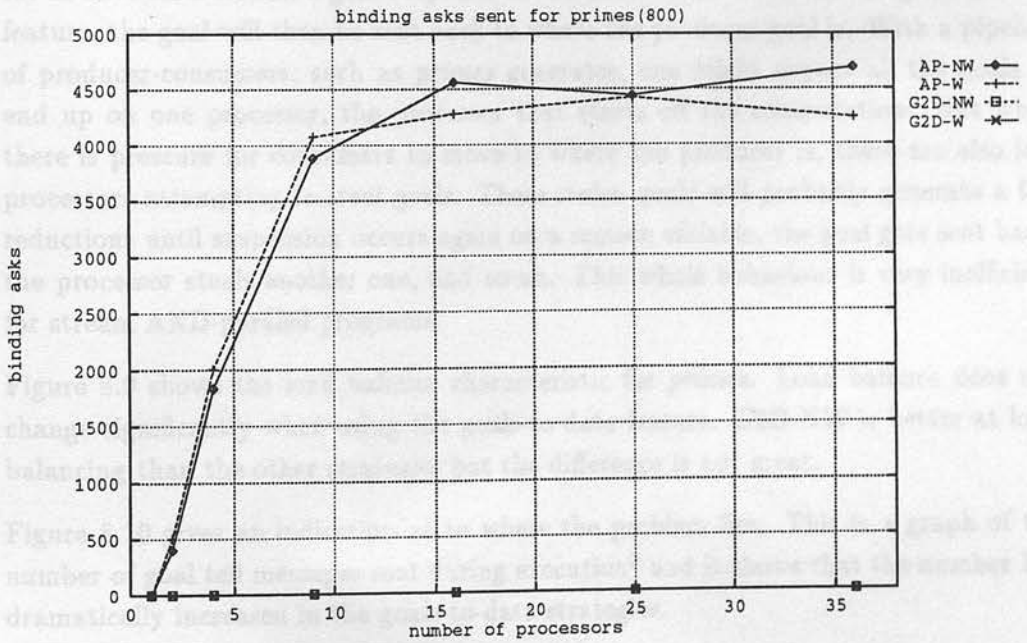

Figure 8.8: Goals-to-data: Binding requests for *primes(800)*

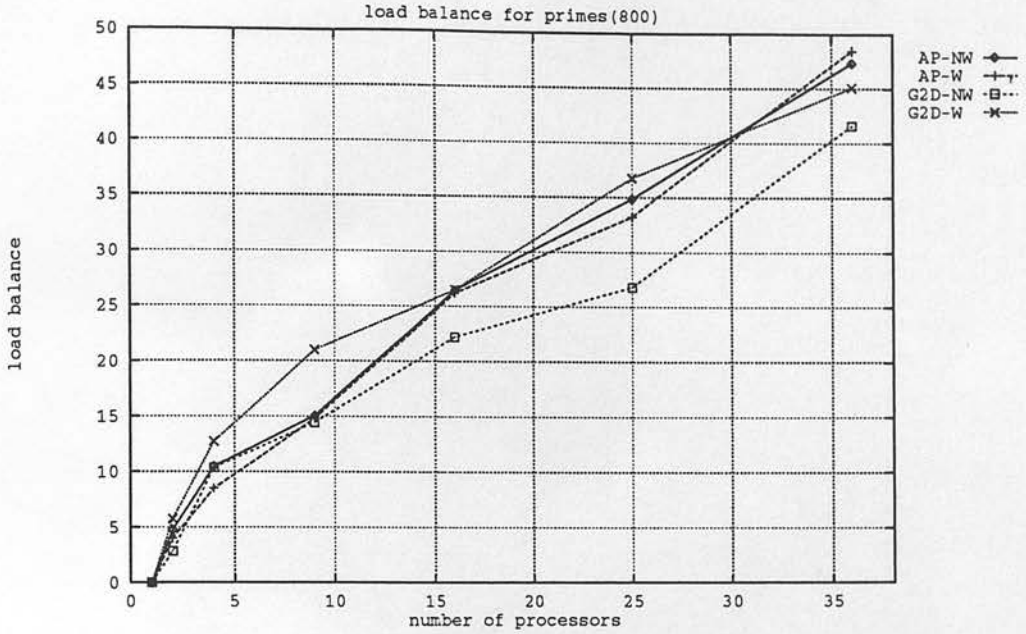

Figure 8.9: Goals-to-data: Load balance for *primes(800)*

To obtain speedup for *primes*, producer and consumer goals must be on separate processors and pass values between each other using shared variables. To request the next set of data on a stream a goal suspends on a remote variable. With the goals-to-data feature, the goal will then be sent back to where the producer goal is. With a pipeline of producer-consumers, such as *primes* generates, one might expect all the goals to end up on one processor, the processor that starts off the computation. But while there is pressure for consumers to move to where the producer is, there are also idle processors attempting to steal goals. These stolen goals will probably generate a few reductions until suspension occurs again on a remote variable, the goal gets sent back, the processor steals another one, and so on. This whole behaviour is very inefficient for stream AND-parallel programs.

Figure 8.9 shows the load balance characteristic for *primes*. Load balance does not change significantly when using the goals-to-data feature. G2D-NW is better at load balancing than the other strategies but the difference is not great.

Figure 8.10 gives an indication as to where the problem lies. This is a graph of the number of goal tell messages sent during execution[2] and it shows that the number has dramatically increased in the goals-to-data strategies.

[2]Goal tell messages are sent whenever a goal is transferred from one processor to another, either in response to a request for a goal from a remote processor or because a goal has suspended on a single variable and the goals-to-data strategy is operating.

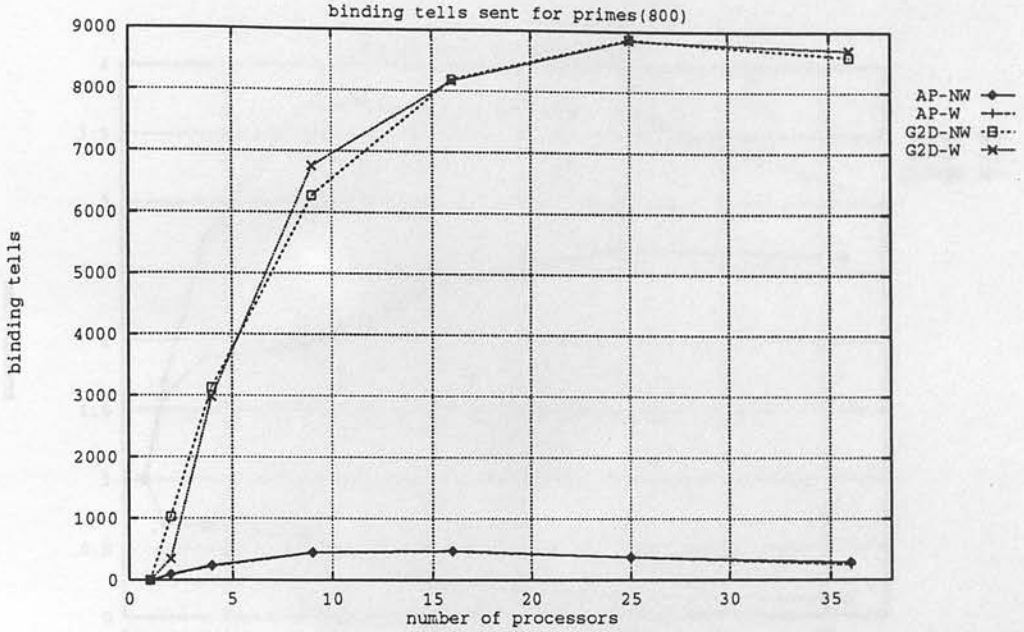

Figure 8.10: Goals-to-data: Goal tells for *primes(800)*

With the addition of the goals-to-data strategy the number of suspensions should have reduced at the expense of sending suspended goals between processors; that is, suspensions should have reduced but goals sent out should have increased by the same amount. But we find that we have a situation where the number of suspensions has slightly increased but that the number of goals sent around the system has increased dramatically: for 25 processors AP-NW shows an average of 370 goal tell messages, whereas G2D-NW shows an average of 8800 goal tell messages, a 24 fold increase!

8.2.1.4 qsort

Only results for G2D-NW were gathered for the *qsort* program.

Figure 8.11 shows the speedup characteristic for *qsort*. Using goals-to-data causes a massive slowdown to a speedup of 0.1. By contrast, without goals to data, AP-NW achieves a peak speedup of 3.5.

Clearly something very costly is happening with the goals-to-data strategy. This is obvious when looking at the load balancing characteristic in figure 8.12. This shows that introducing goals-to-data causes the computation to become extremely unbalanced.

As was conjectured for *primes*, it is likely that goals suspending on remote variables will be sent back to their originating processor, reducing any speedup from stream

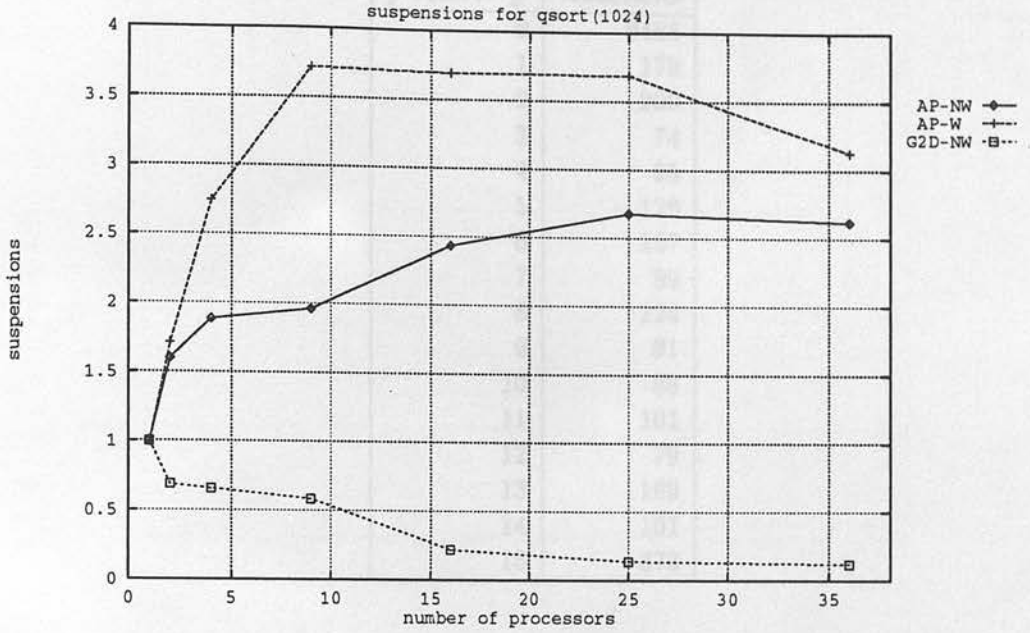

Figure 8.11: Goals-to-data: Speedup for *qsort(1024)*

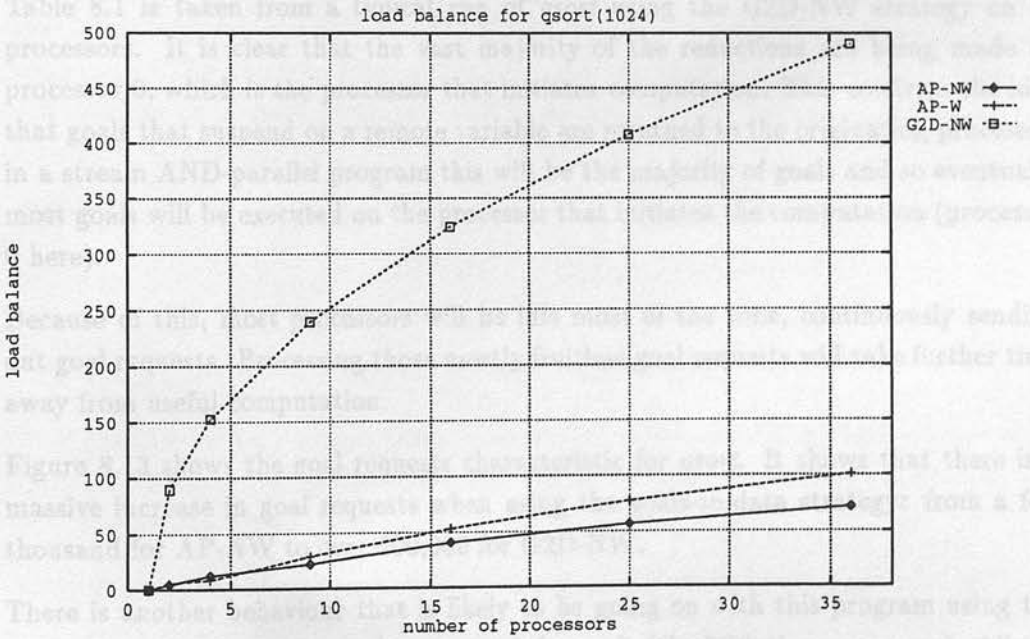

Figure 8.12: Goals-to-data: Load balance for *qsort(1024)*

processor #	*reductions*
0	9485
1	179
2	209
3	74
4	95
5	129
6	157
7	89
8	124
9	91
10	88
11	101
12	79
13	169
14	101
15	373

Table 8.1: Reductions per processor for *qsort(1024)* using G2D-NW on 16 processors

AND-parallelism. In this case it would not be surprising to find all reductions made on one processor.

Table 8.1 is taken from a typical run of *qsort* using the G2D-NW strategy on 16 processors. It is clear that the vast majority of the reductions are being made on processor 0, which is the processor that initiates computation. This confirms the idea that goals that suspend on a remote variable are returned to the originating processor; in a stream AND-parallel program this will be the majority of goals and so eventually most goals will be executed on the processor that initiates the computation (processor 0 here).

Because of this, most processors will be idle most of the time, continuously sending out goal requests. Processing those mostly fruitless goal requests will take further time away from useful computation.

Figure 8.13 shows the goal requests characteristic for *qsort*. It shows that there is a massive increase in goal requests when using the goals-to-data strategy: from a few thousand for AP-NW to over 200,000 for G2D-NW.

There is another behaviour that is likely to be going on with this program using the G2D strategy. One processor has most of the goals (the initiating processor). All the other processors will eventually ask that processor for a goal. The goal executes for a few reductions then attempts to suspend on a remote variable. The goal is sent back to

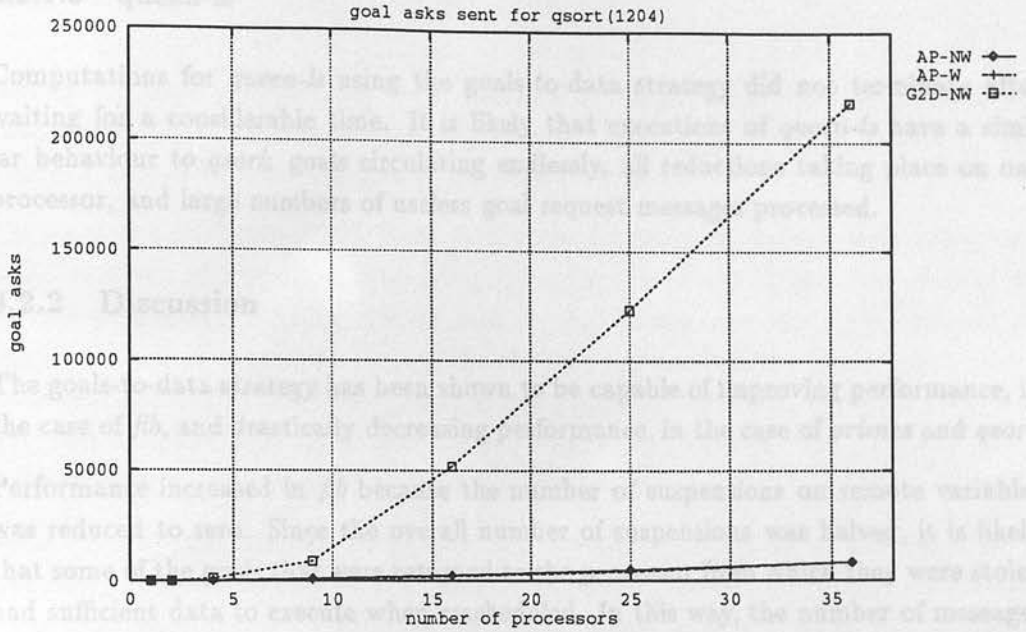

Figure 8.13: Goals-to-data: Goal requests for *qsort(1024)*

the initiating processor. The initiating processor gets many of these goals back when it checks for incoming messages; it will also receive many requests for work from other processors, most of which will be idle. The initiating processor reschedules the goals that it has received, but at the same time it will send out goals on its goal queue to idle processors. The very same goals that were sent back are likely to be sent out again, will attempt to suspend again, get sent back to the initial processor, possibly be sent out again, and so on.

This behaviour would account for some of the slowdown shown in the speedup graph.

In summary:

- Using goals-to-data reduces speedup to almost zero.
- Massive load imbalance was caused using goals-to-data due to most goals being sent back to one processor for evaluation; all other processors were mostly idle during the computation.
- The number of goal requests increases massively when using goals-to-data due to most processors being idle.
- It is possible for goals to circulate around the processors for a long time before being reduced. This may account for much of the slowdown observed.

8.2.1.5 queen-ls

Computations for *queen-ls* using the goals-to-data strategy did not terminate after waiting for a considerable time. It is likely that executions of *queen-ls* have a similar behaviour to *qsort*: goals circulating endlessly, all reductions taking place on one processor, and large numbers of useless goal request messages processed.

8.2.2 Discussion

The goals-to-data strategy has been shown to be capable of improving performance, in the case of *fib*, and drastically decreasing performance, in the case of *primes* and *qsort*.

Performance increased in *fib* because the number of suspensions on remote variables was reduced to zero. Since the overall number of suspensions was halved, it is likely that some of the goals that were returned to the processor from which they were stolen had sufficient data to execute when rescheduled. In this way, the number of messages and suspensions generated was reduced, and performance improved.

For predominantly stream AND-parallel programs, many goals suspend on a remote variable waiting for more data on a stream. This is necessary behaviour to obtain any speedup from the program. Introducing the goals-to-data strategy seriously disrupts that behaviour, sending most goals to execute on one processor (the processor that initiated the computation). At best, this reduces speedup to 1, as was shown with the results from *primes*. At worst, speedup can turn into drastic slowdown, as was the case for *qsort*. This slowdown is likely to be because goals are ping-ponging back-and-forth between the initial processor and the other processors in the system (which are predominantly idle) for a long time before they are reduced. Further analysis would need to be carried out to be certain that this effect is happening.

One method of stopping goals recirculating would be to introduce a new message type `goal_suspend_tell`. Instead of using `goal_tell` to send a goal suspending on a remote variable, a `goal_suspend_tell` would inform the receiving processor that the goal had attempted suspension remotely, and the processor could then put this type of goal on a separate goal queue or mark it as a non-stealable goal. This would stop any recirculating goals, but is still unlikely to improve speedup performance over strategies not using goals-to-data.

In conclusion, the goals-to-data strategy may be useful for programs that have little stream AND-parallelism, but for programs that are predominantly stream AND-parallel using the goals-to-data feature may cause massive performance loss.

8.3 Large numbers of processors

The experiments reported sofar have utilised a maximum of 36 processors. Although this is not an insignificant number of processors, the question might be asked: *what happens when very large numbers of processors are used?*

This has the implication behind it that there is reason to believe that using very large numbers of processors may affect the efficiency of the system. An avenue for further work would be to experiment with the goal distribution strategies executing on large numbers of processors. It has already been shown that the nearest neighbours strategies are bad at distributing load across processors and that the effect (bad load balance) increases with the number of processors. But at least each processor has a fixed number of other processors to obtain goals from.

With the AP strategies, as the number of processors is increased, an idle processor is less likely to find the best goal in the system which may affect system performance. Also if there are many idle processors asking for goals randomly across the computer, the communications network could become overloaded.

8.3.1 Preliminary investigation

The experiments were performed using the emulator executing on 36, 49, 64, 81, and 100 processors.

The queries used in the experiments were *hanoi(22)* (the hanoi program with 22 discs) and *fib(27)*. The problem sizes were increased in order to provide enough work for all the processors; that is, to make sure that speedups were not limited by lack of goals available for computation.

For preliminary experiments, the AP-NW goal distribution strategy was chosen.

Figures 8.14 and 8.15 show the resulting speedup graphs for *hanoi* and *fib* respectively. Each graph has a diagonal line representing linear speedup.

The results for *hanoi(22)* show that efficiency can be very nearly linear with a suitably large computation that has good parallel behaviour: from 1 to 64 processors efficiency is close to 100%, and at 100 processors efficiency is about 85%.

A plot is also shown for *hanoi(15)*, the smaller computation used in previous experiments. It clearly shows that the size of the computation has a large effect on execution efficiency.

The results for *fib(27)* are also encouraging, although the speedup deviates from perfect to a much greater extent than for *hanoi(22)* (because of suspensions caused when trying

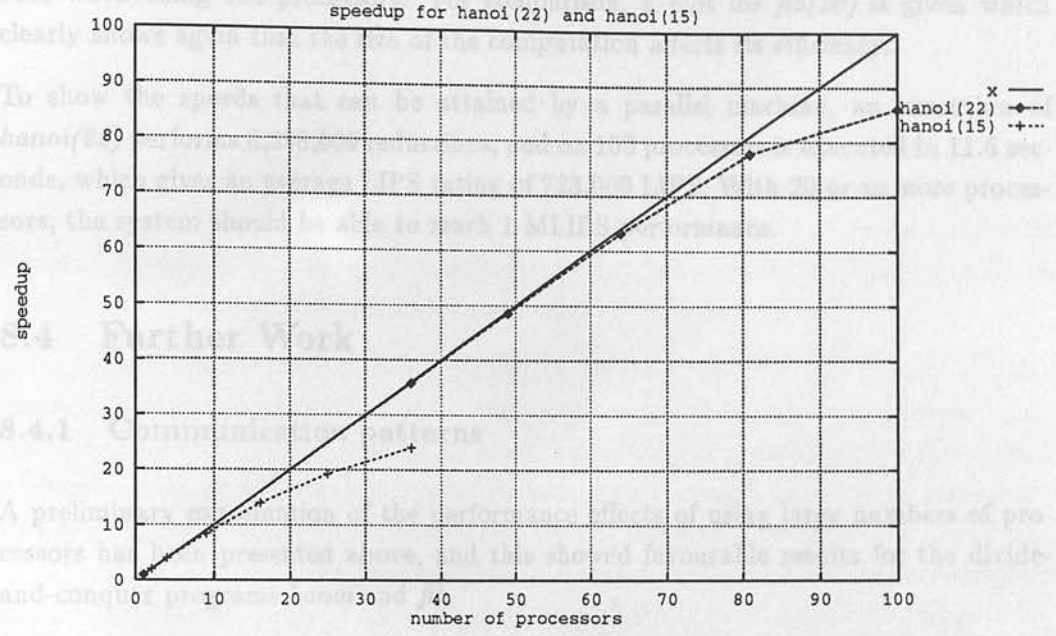

Figure 8.14: Large-processors: Speedup for *hanoi(22)* and *hanoi(15)* using AP-NW

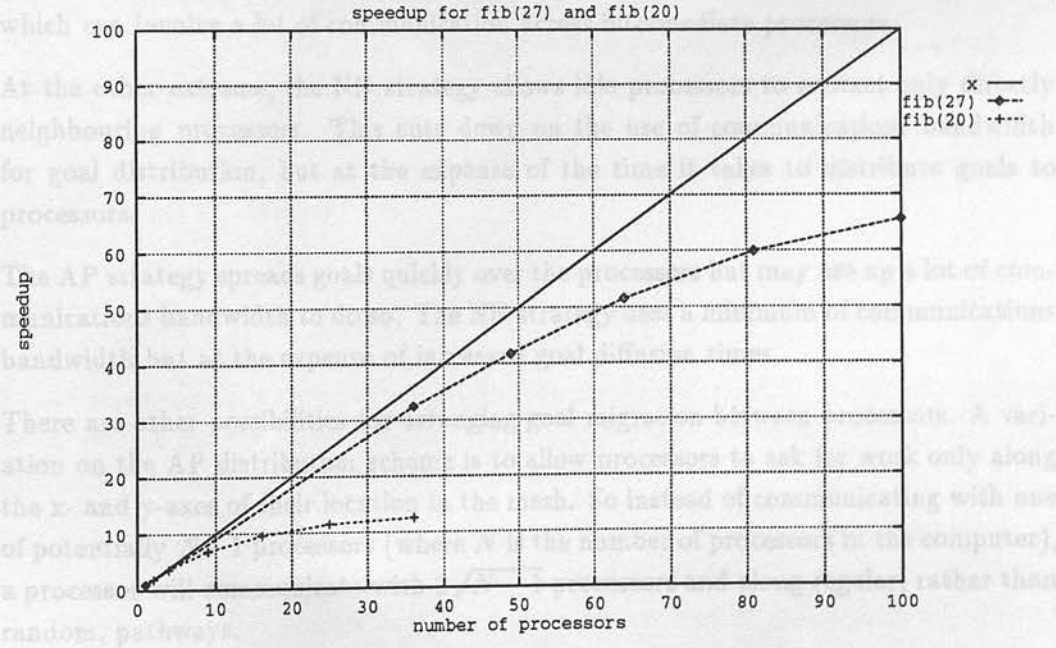

Figure 8.15: Large-processors: Speedup for *fib(27)* and *fib(20)* using AP-NW

to execute *add/3* processes), but efficiency is a high 80% when using 64 processors and 65% when using 100 processors. For comparison, a plot for *fib(20)* is given which clearly shows again that the size of the computation affects its efficiency.

To show the speeds that can be attained by a parallel machine, an execution of *hanoi(22)* performs 8,388,609 reductions, and on 100 processors is executed in 11.6 seconds, which gives an average LIPS rating of 723,000 LIPS. With 20 or so more processors, the system should be able to reach 1 MLIPS performance.

8.4 Further Work

8.4.1 Communication patterns

A preliminary examination of the performance effects of using large numbers of processors has been presented above, and this showed favourable results for the divide-and-conquer programs *hanoi* and *fib*.

A further extension to this, and the other experiments, would be to try different ways of choosing processors to steal work from. The distribution strategies we have looked at so far have been on the extreme ends of the spectrum. The AP strategy allows idle processors to ask any other processor in the computer for work. This means that processors on opposite side of the mesh may try to contact each other to steal work which can involve a lot of communication across intermediate processors.

At the other extreme, the NN strategy allows idle processors to contact only directly neighbouring processors. This cuts down on the use of communications bandwidth for goal distribution, but at the expense of the time it takes to distribute goals to processors.

The AP strategy spreads goals quickly over the processors but may use up a lot of communications bandwidth to do so. The NN strategy uses a minimum of communications bandwidth but at the expense of increased goal diffusion times.

There are other possibilities for arranging goal migration between processors. A variation on the AP distribution scheme is to allow processors to ask for work only along the x- and y-axes of their location in the mesh. So instead of communicating with one of potentially $N-1$ processors (where N is the number of processors in the computer), a processor will communicate with $2\sqrt{N-1}$ processors and along regular, rather than random, pathways.

A variation on the NN distribution scheme might be to allow processors to search for work a certain number of processor hops away. That is, in the current NN strategy the

hop distance is just one processor but this could be increased to two or more processors distances away. Locality of communication is still preserved to an extent while the rate of diffusion of goals across the mesh will improve.

A variation suggested by Steve Gregory (Bristol University) is for an idle processor to seek work from other processors in spiral; that is, first of all the processors that are 1 processor hop away are tried for work, but if none of those provide work then try the processors at 2 processor hops away, and so on. A limit could be put on the maximum distance, in processor hops, that an idle processor can search for work, or it could be allowed to search the entire processor network.

An extension on the goal-to-data distribution strategy would be to prevent the immediate stealing of goals which have been received because they have suspended on another processor. This extension may prevent much of the 'thrashing' of goals between processors that occurs with the simple goals-to-data strategy. One way to implement this extension would be to introduce a new type of message, *goal_suspended(To, From, Goal)*, so that the receiving processor can attempt to suspend **Goal** immediately on receipt instead of placing it on the goal queue where it is in danger of being redistributed.

Bob Fisher (University of Edinburgh) has suggested an alternative method for processors to obtain work. The processors could be arranged in a ring (or 1-D array) and an idle processor would ask the next processor in the ring for work. If that processor had no spare work then the request would be passed on to the next processor in the ring, and so on. This may provide a more systematic way for idle processors to find work than just choosing processors at random. To preserve the desirable properties of the 2-D mesh for distributed unification, the ability to communicate bindings between any two processors quickly, the ring could be implemented as a virtual ring embedded in the physical 2-D mesh.

8.4.2 Granularity control

In our experiments, we conclude that the Tick method of run time granularity analysis using weights did not show significant improvements over scheduling goals in the order in which they appear in the source program.

From the experience gained from researching this thesis, we feel that runtime granularity analysis alone will not be enough to improve efficiency of program execution. It is likely that some compile time analysis will be needed to group goals together to form larger goals, and so increase the grain size of the program.

One of the most promising approaches is that being taken by King at Southampton

University. King's approach to granularity analysis [Kin92] is to analyse the complexity of the goals in a program using abstract interpretation. The goals are classified into one of three complexity groups: constant, linear, and non-linear. Constant and linear complexity goals are not worth distributing to other processors and should be done locally; non-linear complexity goals are worth distributing. There are also dependency rules by which constant and linear goals my be 'merged' with non-linear goals to make larger goals.

Merging goals together is like using the sequential-AND operator ('&') of Parlog between two goals, for example, `A&B`. Instead of creating separate concurrently executing goals for both `A` and `B`, a single goal is created which first executes `A` followed by `B`.

An interesting avenue of further research would be to try King's ideas with our emulator system. What would be needed is that a suitable sequential-AND operator be introduced into FGHC and implemented in the system. Then a number of test programs, having been hand analysed and sequential operators added, could be executed and statistics gathered. This would show the effectiveness of this approach.

Goal distribution strategies will still be needed even using this method since it does not specify how goals should be distributed but only how large goals might be constructed from small goals. Therefore many of the goal distribution strategies will still be relevant to this approach; with the sequential operator defined goals could still be distributed using AP or NN distribution strategies or one of the other alternatives suggested above.

8.4.3 Passing around load information

One conclusion of the experiments in this thesis is that performance is often closely linked to load balancing. This was especially true of the experiments conducted with the interpreter based system.

An extension to our research would be to see if the goal distribution strategies could be improved by having processors exchange various types of load information. Depending on the load information it has about other processors, a processor would be able to make a better choice at choosing a processor that has excess work. In our current AP strategies, idle processors can steal work from lightly loaded processors which may then run out of work themselves, while at the same time there is a heavily loaded processor somewhere else in the network.

There is a tradeoff between the cost of implementing the load exchanging mechanism and the extra performance (if any) gained through its use. As an extreme example, it is obviously very costly to have every processor broadcast full load information to all other processors in a large network of processors because the communications network

would be heavily overloaded, making useful work impossible.

One way of reducing the overhead cost of implementing a load information exchange system, is to piggyback load information onto messages already needed by the system — such as binding ask or goal ask messages.

It would be useful to see if this approach could improve the performance of the blind all-processor distribution mechanism, or whether the extra overhead of processing and acting on load information is counterproductive.

8.4.4 Goal queue orderings

There is a lot of scope for alternative orderings of the goal queue and so different goal distribution strategies.

With the non-weighted strategies investigated in this thesis, goals are executed locally in the order in which they appear in the source code (left-to-right) and are distributed remotely oldest goal first. In other words, the goal queue is used like a stack when executing goals locally but as a queue when distributing goal for remote execution. The idea behind this is to have depth-first local execution of goals but breadth-first distribution of goals.

An alternative way of organising goal queue access might be to take goals for local or remote execution from the same end of the goal queue; that is, the oldest goal is always removed for execution, whether local or remote, and new goals are added to the other end of the queue. This would give breadth-first local goal execution and breadth-first goal distribution. It is not obvious whether this would be a better approach than that taken in this thesis.

Steve Gregory has pointed out that a more efficient way of implementing the weighted queue used in the weighted strategies may be to not keep the queue ordered by weight at all. Instead a search of the goal queue could be made for the goal with largest weights at the time the *goal_ask* message is received. This would save the time taken merge sorting each goal into the goal queue but would take more time to find the goal with greatest weight after a remote goal request. Whether this approach will reduce the time spent in managing the goal queue for weights depends on the cost of searching the queue, the cost of merge sorting a goal, and the frequencies with which both operations occur in the different implementations.

One aspect of goal ordering that has had little consideration in our implementations is that of **fairness**. A fair system will guarantee to attempt reduction on each goal in the goal queue at some time, even if some of the goals require infinite time to execute. Some goal orderings are unfair: for example, on one processor, if the first goal to be

executed in a local depth-first execution system is infinite then none of the other goals will be executed. The simple AP strategy (local goal execution is depth-first, remote goal execution is breadth-first) is not guaranteed to be fair, but the variation suggested by Steve Gregory above would be fair.

8.4.5 More ideas

more programs In experiments described in this thesis, the behaviour of five benchmark programs has been explored. It would be as well to execute a much larger range of programs over the systems to see how they behave and to get a much broader feel for how the various types of goal distribution strategy affect program execution.

As well as benchmark programs, a range of large 'real' applications programs should be executed and the effectiveness of the goal distribution strategies for these programs evaluated.

pathological programs Instead of executing 'standard' benchmark programs, executing synthetic programs that are designed to exhibit pathological behaviours may help to show effects of goal distribution strategies. (*hanoi* can be viewed as a pathologically good program.) As examples, consider programs that create vast numbers of small goals, create many goals that communicate bindings from one to another in a pipelined or tree-like fashion, goals that create many bindings or that generate large amounts of communication traffic. By trying to invent distribution strategies to cope with such pathological programs we might better understand how to cope with more usual program types.

profiling tool A profiling tool, similar to the one designed by Trehan [Tre89], could help the researcher to understanding the effects of goal distribution strategies. Trehan collects various information, such as number of suspensions and reductions made per time tick in a simulated execution of programs notionally executing on an infinite number of processors with no communication cost for sending a message. Using the information gathered at each time tick, graphs are plotted for the number of suspensions or reductions performed against time.

The information gathered from the systems described in this thesis is based on counters, and only describes the result of the whole computation; for example, total number of reductions and suspensions made on each processor. It is not possible to get a feel of how the computation progresses over time.

A simple way to use Trehan's ideas would be to periodically save the counters used to collect the program statistics (such as reductions made and messages sent) together with a timestamp of the elapsed time from the start of program execution, giving some indications of the state of a processor over time.

This would also be an advancement of Trehan's work since it would generate plots of programs executing in a real, rather than simulated, parallel environment. The results gathered from such experiments could be compared with his work on simulated executions.

8.5 Summary

This chapter started by considering a modification to the goal distribution strategies where goals that attempt to suspend on a single remote variable are instead sent to the processor to which the variable refers. This strategy is called the **goals-to-data** strategy.

The results of experiments for the benchmark programs using the AP strategies with and without the goals-to-data feature were presented. It was shown that for some programs at least (*fib*) using goals-to-data can provide some performance enhancement. This is achieved by reducing the number of binding requests and suspensions generated and by improving load balancing. This effect was noticed only with the non-weighted strategy; the strategy ordering goals by weights showed no performance difference when using goals to data.

For highly stream AND-parallel programs (*qsort* and *primes*) a substantial performance loss was noticed when using goals-to-data. This was found to be due to most goals being reduced on a single processor, the processor that initiates the computation. With stream AND-parallel programs, many goals suspend waiting for further values on streams. Instead of suspending on the local processor, most goals will be sent back to the processor from which they were stolen in the first place. Eventually, all goals will finish up on the same processor and hence performance (speedup) will be totally undermined.

The next part of the chapter considered what might happen if the number of processors was increased and large numbers of processors used. Experiments with up to 100 processors showed that acceptable performance could be attained for large computations of the *hanoi* and *fib* programs. The experiments also underlined the effect that computation size has on performance: a large computation is more likely to show good speedup characteristics than a small computation.

Finally, ideas for further work were presented.

Chapter 9

Conclusions

The main contributions of the work described in this thesis are presented followed by a summary of conclusions.

9.1 Contributions

The contributions fall into two categories: firstly, the design and implementation of tools for exploring goal distribution and scheduling ideas; and secondly, using the tools, the performance analysis of various goal distribution and scheduling strategies.

9.1.1 Tools

9.1.1.1 Language implementations

Two implementations of Flat Guarded Horn Clauses—an interpreter and a WAM-like system—were designed and implemented. They were designed to execute on commercially available distributed memory computers.

The implementations were executed on a Meiko T800 Transputer array using C with CSTools message passing extensions. All that would be needed to use the systems with other computers is to re-implement the message passing routines, contained in the Communications Manager module, in the toolset used by the computer. Both the interpreter and emulator have also been executed on a network of Sun SPARC machines, using CSTools communications libraries.

The emulator compares favourably the main comparable commercial implementation of a committed-choice logic language, Strand88.

There are very few parallel implementations of the committed-choice languages, especially those which execute on commercial hardware. Providing these implementations, and describing their design, adds to the so far limited practical knowledge in this field.

Related committed-choice language implementations are: Crammond's [Cra88] work on implementations for shared memory computers; Taylor et al. [TSS87] work on implementing Flat Concurrent Prolog for distributed memory machines; and the implementation of Strand88 [FT90] for a variety of computers. There are also a number of implementations of FGHC arising from the Japanese Fifth Generation Project [IMT87], but they have been designed to execute on proprietary hardware.

9.1.1.2 Measures

A set of measures has been defined for characterising the performance of a committed-choice language system executing on a distributed memory machine. These measures have proved useful for gaining knowledge of the behaviour of parallel execution of programs and the effects of goal distribution strategies.

The measures are:

Average execution time: one of the main reasons for using a parallel computer is to minimise execution time.

Average speedup: this shows how well processors are utilised. A good system will maximise speedup.

Load balance: defined in this thesis as the average coefficient of variance of the reductions across the processors. A good system will to minimise this measure.

Average suspensions: this shows the number of goals that suspend during program execution. Suspending a goal takes time and so a good strategy may try to minimise this measure.

Average number of binding requests: this indicates the number of goals suspending on remote variables. A good system strives to minimise this measure, as long as speedup is not reduced.

Average number of goal requests: this is an indicator of the idleness of processors. A good system strives to minimise this measure.

9.1.1.3 Goal distribution model

We have defined a model for describing goal distribution and scheduling strategies based on access functions to the goal queue. This is a formalisation of the methods that other researchers have used, most notably Crammond [Cra90]. A formalised goal queue access functions makes explicit the goal distribution strategies supported by that model. Varying the access functions to the goal queue allows different scheduling and distribution strategies to be explored.

Another aspect that the goal distribution model makes explicit is that, in a demand driven system, a procedure is specified for choosing a processor to ask for spare work. Defining this procedure in different ways produces variations in goal distribution strategy.

9.1.2 Evaluation of strategies

Using the above tools, a number of goal distribution and scheduling strategies were designed and implemented and their performance analysed. The strategies investigated have two dimensions: all-processors (AP) versus nearest-neighbours (NN), and non-weighted versus weighted. Preliminary investigations were also made into two other effects: migrating suspended goals to data, and using large number of processors.

9.1.2.1 Weights or no-weights

The non-weighted strategy orders the goal queue such that goals executed are ordered locally so that program execution is depth-first, and goals are distributed in a breadth-first manner. The motivation to use this strategy was that it is the very simple and requires no user involvement in the goal distribution process. Crammond used this strategy [Cra90], and variations on it, for his shared memory implementations. We are not aware that any other detailed empirical analysis has been made for this distribution strategy for distributed memory computers.

The weighted-strategy orders goals by weight, calculated from the source code using heuristics. The motivation to use this strategy was that it might be a simple way to improve system performance. Goals with large weight should be distributed first to keep remote processors busy for as long as possible, and that it is more efficient to execute goals that generate little work locally rather than remotely. This strategy was proposed by Tick [Tic90] and some preliminary empirical analysis was made for it with a shared memory implementation of FGHC. As far was we know, no empirical analysis has been made for this strategy using distributed memory implementations of

committed-choice languages.

9.1.2.2 All-processors or nearest-neighbours

The all-processors/nearest-neighbours dimension relates to which processor an idle processor should choose to obtain more work. All-processors allows an idle processor to steal work from any other processor; nearest-neighbours allows an idle processor to steal work only from neighbouring processors.

The AP strategy was again based on work by Crammond [Cra90]. In his shared memory implementation he allows idle processors to steal goals from the goal queue of any other processor. Using such a strategy incurs little overhead in a shared memory implementation because all the goal queues are stored as shared objects; removing a goal from another processor's goal queue takes about the same time as removing a goal from the idle processor's own goal queue. In a distributed memory environment the cost of accessing a remote goal queue is much higher than accessing the local goal queue. The motivation to use this strategy was to see if it was still viable in a distributed memory environment.

The NN strategy takes the opposite view to the AP strategy. The motivation to test this strategy was to see if preserving some locality of data and goals would improve system performance.

The two dimensions give four possible goal distribution strategies:

AP-NW	All-Processors Non-Weighted;
AP-W	All-Processors Weighted;
NN-NW	Nearest-Neighbour Non-Weighted;
NN-W	Nearest-Neighbour Weighted.

The results of executing benchmark programs on our FGHC systems using these strategies were collected and analysed using the measures. Through this, knowledge has been gained of how the goal distribution strategies behave and perform, how different types of program behave with different strategies, and how useful the measures were for predicting and describing the behaviour of system executions.

9.1.2.3 Goals-to-data

Using a goals-to-data strategy involves sending goals that suspend on a single remote variable to the processor on which that processor resides. It can be used in conjunction with any of the distribution strategies described above. The motivation behind this

modification was to see if the numbers of suspensions and binding requests messages would be reduced, increasing system performance. Empirical results were gathered for the AP strategies using the goals-to-data modification. To our knowledge, this strategy has not been tried before.

9.1.2.4 Large numbers of processors

Performance of the AP-NW strategy as the number of processors is increased to 100 processors was investigated. The motivation was to see if the AP-NW strategy is still useful for such large numbers of processors or whether inefficiencies due to many processors stealing work from each other hamper system performance.

9.2 Conclusions

The conclusions are segmented as to the different experiments made. Conclusions from the interpreter experiments are presented first, followed by those from the emulator experiments. This section finishes with conclusions from the goal-to-data experiments and the experiments made with large numbers of processors.

9.2.1 Interpreter experiments

Load balance was the best indicator of goal distribution strategy performance. The other measures — suspensions, binding asks and goal requests — did not significantly correlate with execution time.

The AP strategies are better than the NN strategies because they are consistently better at load balancing. The NN strategies are worse at load balancing because the time taken for goals to diffuse across processors is longer than for the AP strategies.

Ordering goals by weight has a varying effect on program performance. Four out of five of the test programs showed either no or very little performance increase when using a weighted strategy.

Execution of *fib* showed that keeping small goals that immediately suspend locally is a better strategy in the long term than sending them out to suspend remotely.

For predominantly stream AND-parallel programs to obtain any speedup producer and consumer goals must execute on different processors. Such an execution mode generates many suspensions and binding requests due to consumers suspending waiting for more data from producers. Good execution of stream AND-parallel programs therefore

generates many suspensions and binding requests. For this reason, for stream AND-parallel programs, large number of suspensions and binding requests indicates good performance.

Overall, AP-NW is the best goal distribution strategy.

9.2.2 Emulator experiments

We hypothesised that the interpreter spend most of its time reducing goals compared to the time spent in suspending processes and sending messages, and that this may account for the lack of a negative effect on execution time due to suspensions and message processing.

To test this theory we implemented a WAM-like emulator, derived from the interpreter, but where the time taken to perform a reduction was greatly reduced; all other aspects of the system are identical to the interpreter. The same experiments were then performed on the emulator as were performed with the interpreter.

The emulator system executed programs between 6 and 8 times faster than the interpreter when measured on a single processor, confirming the view that the average reductions time is inflated in the interpreter.

As with the interpreter, the AP strategies are better at load balancing than the NN strategies.

It is less clear than in the case for the interpreter experiments which goal distribution strategy gives the best execution time. Average suspensions has become the best predictor of execution time for the emulator; with the interpreter it was load balancing. This is likely to be because of the reduced effect of reduction time making suspensions more prominent coupled with the fact that the load balancing of the emulator experiments were worse than those for the interpreter; that is, load balancing has become harder and probably less significant.

There was a noticeable worsening of speedup characteristic when comparing the emulator to the interpreter. This is because overheads such as handling suspensions and messages have become more prominent compared to time spent reducing goals.

Using NN strategies can sometimes improve performance. This is thought to be due to the slow diffusion rate of goals keeping small goals on the same processor (improving locality). Instead of these small goals being executed remotely inefficiently, they are executed locally, reducing suspensions and binding requests.

9.2.3 Goals-to-data

A preliminary investigation into the effect of adding an extra rule to the AP strategies — the goals-to-data rule — was undertaken. The goals-to-data rule is:

> *A goal that suspends on one remote variable only is sent to the processor that owns that variable.*

The hope was that this would reduce the number of goals suspended on remote variables and so reduce the suspension count and improve execution performance.

In general, the AP strategies **without** goals-to-data perform better than those with the extra rule. The exception to this was *fib* where G2D-NW gave the least execution time due to reducing the number of suspensions and binding requests and improving load balancing.

For stream AND-parallel programs, using goals-to-data reduces speedup to 1 at best. This is because consumer goals are sent to the processor where their producer goal is executing. If the program consists of mostly producer-consumer goals, then eventually all goals will execute on a single processor, hence a speedup of 1.

At worst, stream AND-parallel programs may experience dramatic slowdown when using the goals-to-data strategy. This may be due to goals recirculating between processors: they attempt to suspend on a variable, are sent to the variable's owning processor, are stolen again by an idle processor, suspend again and so on.

The main conclusion is that this is not a promising strategy: it produces slowdown in many programs.

9.2.4 Large numbers of processors

The performance of AP-NW, was very good with the number of processors we tried — from 36 up to 100.

Another feature that was noticed was that size of computation affects performance, large computations being more efficient than small computations.

The AP strategy remains promising even when using very large numbers of processors.

9.2.5 Overall conclusions

The AP strategy looks very promising. This strategy is also likely to improve with the trend to implement wormhole routing in hardware, as in the new Inmos T9000 series

Transputers, as the message times between any two processors becomes more similar. The encouraging results shown by this strategy give us hope that there are reasonable automatic goal distribution strategies and that users will not have to resort to user generated goal distribution annotations.

The main worry is that employing large numbers of processors might reduce the effectiveness of this strategy, although our experiments so far have failed to find significant problems with the AP strategy with up to 100 processors. It still remains to be seem whether this strategy can sustain numbers of processors into the hundreds or thousands.

The NN strategy does not look promising. It imposes a maximum diffusion rate at which goals can move across the machine and has a bad load balancing characteristic, which becomes worse as the number of processors is increased.

Using weights does not have a consistent effect on system performance and where weights does make a difference it is generally small. Ordering goals by static weights does not look like a promising strategy although research on a more varied range of programs is needed.

The goals-to-data strategy does not look useful. For predominantly stream AND-parallel programs, which many committed-choice programs are likely to be, it can cause substantial slowdown.

Comparing the different program styles tested — independent AND-parallel versus stream AND-parallel — it is clear that the programs with largely independent AND-parallel show better speedups than stream AND-parallel programs. Clearly, employing as much independent AND-parallelism in a program is desirable, as is avoiding the use of stream-AND parallelism if possible.

To summarise the conclusions to this thesis in one sentence: to obtain the best speedup when choosing one of our automatic goal distribution strategies, use the all-processors no-weights (AP-NW) strategy and use as much independent AND-parallelism in programs as possible.

Bibliography

[Ait90] Ait-Kaci, H. The WAM: A (Real) Tutorial. Technical Report 5, Paris Research Laboratory, DEC, Paris, France, 1990.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 3rd edition, 1987.

[Cra88] J. Crammond. Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors. Technical Report PAR 88/4, Department of Computing, Imperial College, London, 1988.

[Cra90] J. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 642–657, Cambridge, MA, 1990. MIT Press.

[CS87] M. Codish and E. Shapiro. Compiling OR-Parallelism into AND-parallelism. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, chapter 32, pages 351–382. MIT Press, 1987.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, 1984.

[Fly72] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computing*, C-21:948–960, 1972.

[Fos90] I. Foster. Automatic Generation of Self-Scheduling Programs. Technical Report IL 60439, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, 1990.

[FT88] I. Foster and S. Taylor. Flat Parlog: A Basis for comparison. *International Journal on Parallel Programming*, 16(2), 1988.

[FT90] I. Foster and S. Taylor. *Strand: new concepts in parallel programming.* Prentice-Hall, 1990.

[GDBD92] D. Gudeman, K. De Bosschere, and S. K. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *1992 Joint Conference and Symposium on Logic Programming*, 1992.

[Gre87] S. Gregory. *Parallel logic programming in PARLOG: The language and its implementation.* Addison-Wesley, 1987.

[HKTT91] Y. Hidaka, H. Koike, J. Tatemura, and H. Tanaka. A Static Load Partitioning Method Based on Execution Profile for Committed Choice Languages. In *International Conference on Logic Programming 1991*, 1991.

[HS87] A. Houri and E. Shapiro. A Sequential Abstract Machine for Flat Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: collected papers*, volume 2, chapter 38, pages 513–574. MIT Press, 1987.

[IMT87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Fourth International Conference on Logic Programming*, pages 257–275, Melbourne, 1987. MIT Press.

[KC87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *1987 Symposium on Logic Programming*, 1987.

[Kin92] A. King. *Compile-time Analysis of Concurrent Logic Programs for Multiprocessors.* PhD thesis, Department of Electronics and Computer Science, University of Southampton, 1992.

[MW88] D. Maier and D. S. Warren. *Computing with Logic.* Benjamin/Cummings, 1988.

[NT88] M. Nilsson and H. Tanaka. Massively Parallel Implementation of Flat GHC on the Connection Machine. In *International Conference on Fifth Generation Computer Systems*, pages 1031–1040, 1988.

[OM87] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *1987 Symposium on Logic Programming*, pages 224–232, San Fransisco, 1987. IEEE Computer Society Press.

[Sar89] V. A. Saraswat. *Concurrent constraint programming languages.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1989.

[Sha87a] E. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In E. Shapiro, editor, *Concurrent Prolog: collected papers*, volume 1, chapter 2, pages 27–83. MIT Press, 1987.

[Sha87b] E. Shapiro. Concurrent Prolog: A Progress Report. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, chapter 5, pages 156–187. MIT Press, 1987.

[Sha87c] E. Shapiro. Systolic Programming: A Paradigm of Parallel Processing. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, chapter 7, pages 207–242. MIT Press, 1987.

[SKL90] V. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In *1990 North American Conference on Logic Programming*, pages 431–446. MIT Press, 1990.

[TARS87] S. Taylor, A. Av-Ron, and E. Shapiro. A Layered Method for Process and Code Mapping. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, chapter 22, pages 79–100. MIT Press, 1987.

[Tic90] E. Tick. Compile-time granularity analysis for parallel logic programming languages. *New Generation Computing*, 7:325–337, 1990.

[Tre89] R. Trehan. *An Investigation of Design and Execution Alternatives for the Committed Choice Non-Deterministic Logic Languages.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1989.

[TSS87] S. Taylor, S. Safra, and E. Shapiro. A Parallel Implementation of Flat Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, chapter 39, pages 575–604. MIT Press, 1987.

[Uch83] S. Uchida. Toward a New Generation Computer Architecture: Research and Development Plan for Computer Architecture in the Fifth Generation Project. Technical Report TR-001, ICOT, Tokyo, Japan, 1983.

[Ued86] K. Ueda. Guarded horn clauses: A parallel programming language with the concept of a guard. Technical Report TR-208, ICOT, 1986.

[Ver91] A. R. Verden. *And-Parallel Implementation of Prolog on Distributed Memory Machines.* PhD thesis, Department of Electronics and Computer Science, University of Southampton, 1991.

[War83] Warren, D. H. D. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, 1983.

Appendix A

Benchmark Programs

The benchmark programs used in the experiments are listed here.

A.1 nrev

```
main :- go(150).

append([], L, R) :- R=L.
append([H|T], L, R) :- R=[H|R1], append(T, L, R1).

nrev([], R) :- R=[].
nrev([H|T], R) :- nrev(T, NT), append(NT, [H], R).

go(150) :- nrev([1,2,3,4,5,...,150], _).
```

This is a standard program for naive reverse. The query reverses a list of 150 integers.

A.2 hanoi

```
:- weight(hanoi/1,2).
:- weight(move/4, 2).

main :- hanoi(15).

hanoi(N) :- move(N, left, center, right).

move(0, _, _, _).
move(N, A, B, C) :- N =\= 0, M is N-1 :
  move(M,A,C,B),
  move(M,C,B,A).
```

This a translation of a Prolog program from [Ver91]. When translated into FGHC the Prolog ordering of goals in the body of the 2nd clause of *move/4* is not preserved, so the FGHC implementation is not a true implementation of the *hanoi* problem. It does, however, make a good pure divide-and-conquer test program: its execution results in a balanced tree of independently executing *move/4* goals.

A.3 fib

```
:- weight(fib/2, 2).
:- weight(add/3, 0).

main :- fib(20, _).

fib(0, R) :- true : R = 1.
fib(1, R) :- true : R = 1.
fib(N, R) :- N>1, N1 is N-1, N2 is N-2 :
  fib(N1, R1), fib(N2, R2), add(R1, R2, R).

add(R1, R2, R) :- S is R1+R2 : R=S.
```

This is another program translated from a Prolog version from [Ver91]. It is a program to recursively calculate Fibonacci numbers. On execution an unbalanced tree of *fib/2* goals is created. Each pair of *fib/2* goals has a dependent *add/3* goal. This program was chosen because it introduces a small amount of dependency into an otherwise highly independent AND-parallel program and the effect of that dependency can be studied.

Notice that since *add/3* reduced to a system goal it has a weight of zero. (From the strict application of Tick's weight definitions it would have a weight of zero in any case.)

A.4 qsort

```
:- weight(go1024,    3).
:- weight(qsort/3,   3).
:- weight(part/4,    1).
:- weight(list256/2, 0).

main:- go1024(_).

qsort([],Rest,Ans) :- true : Rest=Ans.
qsort([X|R],Y,T)   :-
  part(R,X,S,L), qsort(S,Y,[X|Y1]), qsort(L,Y1,T).

part([X|Xs],A,S,L) :- A<X  : L=[X|L1], part(Xs,A,S,L1).
part([X|Xs],A,S,L) :- A>=X : S=[X|S1], part(Xs,A,S1,L).
part([]    ,_,S,L) :- true : S=[], L=[].

go1024(_) :- list256(L,L2),list256(L2,L3),
             list256(L3,L4),list256(L4,[]),
             qsort(L,A,[]).

list256(L,E) :-
  L =
    [128,64,192,32,96,160,224,16,48,80,112,144,176,208,240,8,
      24,40,56,72,88,104,120,136,152,168,184,200,216,232,248,4,
      12,20,28,36,44,52,60,68,76,84,92,100,108,116,124,132,
     140,148,156,164,172,180,188,196,204,212,220,228,236,244,252,2,
       6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,66,
      70,74,78,82,86,90,94,98,102,106,110,114,118,122,126,130,
     134,138,142,146,150,154,158,162,166,170,174,178,182,186,190,194,
     198,202,206,210,214,218,222,226,230,234,238,242,246,250,254,1,
       3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,
      35,37,39,41,43,45,47,49,51,53,55,57,59,61,63,65,
      67,69,71,73,75,77,79,81,83,85,256,87,89,91,93,95,
      97,99,101,103,105,107,109,111,113,115,117,119,121,123,125,127,
     129,131,133,135,137,139,141,143,145,147,149,151,153,155,157,159,
     161,163,165,167,169,171,173,175,177,179,181,183,185,187,189,191,
     193,195,197,199,201,203,205,207,209,211,213,215,217,219,221,223,
     225,227,229,231,233,235,237,239,241,243,245,247,249,251,253,255|E].
```

This program is a difference list implementation of quicksort. It has been borrowed from an anthology of programs compiled by Tick. The standard execution is to sort a list of 1024 integers, constructed from four identical lists of 256 integers. Execution of the program generates an AND-tree of *part/4* goals. The shape of the tree depends on the data sorted.

A.5 primes

```
:- perpetual gen/3.

:- weight(primes/2, 2), weight(sift/2,   2).
:- weight(filter/3, 1), weight(filter/5, 1).

main :- primes(800,_).

primes(S,X) :- true : gen(2,S,A), sift(A, X).

gen(Max, Max, Out) :- true : Out = [].
gen(N, Max, Out) :-  N < Max, N1 is N + 1 :
  Out = [N|Out1],
  gen(N1, Max,Out1).

sift([], R) :- true : R=[].
sift([H|T], R) :- true :
  R = [H|R2],
  filter(H, T, R1),
  sift(R1, R2).

filter(_, [], R) :- true : R = [].
filter(P,[H|T], R):- M is H mod P : filter(M, P, H, T, R).

filter(0, P, H, T, R) :- true : filter(P, T, R).
filter(M, P, H, T, R) :- M =\= 0 :
  R = [H|R1], filter(P, T, R1).
```

This is a well known program for calculating prime numbers using the generate-and-test paradigm. On execution a *gen/3* goal generates a stream of integers starting from 2. This stream is consumed by a *filter/5* goals, one for each prime number. They filter out multiples of their prime.

Execution of the program generates a pipeline of filter goals. Filter goals with low primes are more active than those with high primes. Most filter goals will be suspended most of the time.

Notice that *gen/3* is a simple recursive goal and has been made perpetual to give it a weight of zero.

A.6 queen-ls

```
:- perpetual count/2, count/3.

:- weight(go/3,       50), weight(queen/4,    35).
:- weight(q/4,        25), weight(filter/4,   15).
:- weight(fromLStoL/2,20), weight(fromLStoS/4,20).
:- weight(count/2,     0), weight(count/3,     0).

main :- go(8,_,_).

go(M,N,A) :- true :
  queen(1,M,begin,A), fromLStoL(A,B), count(B,N).

queen(I,N,In,Out) :- I =< N, I1 := I+1 :
  q(1,N,In,In1),
  queen(I1,N,In1,Out).
queen(I,N,In,Out) :- I > N : Out = In.

q(I,N,In,Out) :- I =< N, I1 := I+1 :
  Out = [[I|In1]|Out1],
  filter1(In,I,1,In1),
  q(I1,N,In,Out1).
q(I,N,_,Out) :- I > N : Out = [].

filter1([[J|In]|Ins],I,D,Out) :-
  J =\= I, D =\= I-J, D =\= J-I, D1 := D+1 :
  Out = [[J|NewIn]|Out1],
  filter1(In,I,D1,NewIn),
  filter1(Ins,I,D,Out1).
filter1(begin,_,_,Out) :- true : Out=begin.
filter1([],   _,_,Out) :- true : Out=[].
otherwise.
filter1([[_|_]|Ins],I,D,Out) :- true :
  filter1(Ins, I, D, Out).

fromLStoL(LayeredStream,List) :- true :
  fromLStoS(LayeredStream,[],List,[]).

fromLStoS([[A|LS1]|Rest],Stack,L0,L2) :- true :
  fromLStoS(LS1,[A|Stack],   L0,L1),
  fromLStoS(Rest,Stack,      L1,L2).
fromLStoS(begin,Stack,L0,L1) :- true : L0=[Stack|L1].
fromLStoS([],_,L0,L1) :- true : L0=L1.

count(L,N) :- true : count(L,0,N).

count([X|Xs],M,N) :- M1 := M+1 : count(Xs,M1,N).
count([],M,N) :- true : N = M.
```

This is a program for generating all-solutions to the N-queens problem — how to place N queens on an N-by-N chess board such that no queen attacks any other — using the layered streams technique [IMT87]. The layered stream is flattened down to a single stream by *fromLStoS/4* and the solutions are counted by *count/3*. The *count/3* goal is simply recursive and has been declared as perpetual with a weight of zero. When calculating weights it was found that fractional weights result (for example 3.5), so they were multiplied by ten to give integer values.

This program is one from an anthology collected by Tick.